Java Persistence et Hibernate

Anthony Patricio

Avec la contribution de Olivier Salvatori

ISBN : 978-2-212-12259-6

EYROLLES

2

Démarrer avec Java Persistence

Le rôle de Java Persistence et des outils de mapping objet-relationnel en général est de faire abstraction du schéma relationnel qui sert à stocker les entités. Les métadonnées permettent de définir le lien entre votre conception orientée objet et le schéma de base de données relationnelles. Avec Java Persistence, le développeur d'application n'a plus à se soucier *a priori* de la base de données. Il lui suffit d'utiliser l'API Java Persistence, notamment l'API `EntityManager`, et les interfaces de récupération d'objets persistants.

Les développeurs qui ont l'habitude de JDBC et des ordres SQL tels que select, insert, update et delete doivent changer leur façon de raisonner au profit du cycle de vie de l'entité. Depuis la naissance d'une nouvelle instance persistante jusqu'à sa mort en passant par ses diverses évolutions, ils doivent considérer ces étapes comme des éléments du cycle de vie d'un objet et non comme des actions SQL menées sur une base de données.

Toute évolution d'une étape à une autre du cycle de vie d'une entité passe par le gestionnaire d'entités. Ce chapitre introduit les éléments constitutifs d'un gestionnaire d'entités.

Avant de découvrir les actions que vous pouvez mener sur des entités depuis le gestionnaire d'entités et leur impact sur la vie des objets, vous commencerez par créer un projet fondé sur JBoss intégré afin de pouvoir tester les exemples de code fournis.

Mise en place

Cette partie en apparence laborieuse est cruciale. Elle permet de mettre en place un socle d'exécution des exemples qui illustrent ce livre. Une fois l'installation effectuée, ce socle se montrera des plus complet et facile à manipuler. Il vous permettra, au-delà de Java Persistence, d'aborder et comprendre d'autres standards et couches techniques en rapport avec la problématique traitée.

Lors de l'écriture de mon ouvrage précédent, *Hibernate 3.0, Gestion optimale de la persistance dans les applications Java/J2EE*, il était cohérent de partir sur des modèles autonomes, sans conteneur, faciles à modifier et à exécuter. Le choix de JUnit en mode autonome (*standalone*, SE) s'imposait de lui-même puisqu'il y avait peu d'intérêt à exécuter les exemples sur la plate-forme Entreprise (EE).

Avec Java Persistence, il en va autrement. D'abord parce que Java Persistence spécifie le standard de déploiement et qu'il nous faut donc exécuter nos exemples sur une plate-forme capable d'exploiter ce standard. Ensuite parce qu'il y a plusieurs composants de la plate-forme Entreprise qu'il est intéressant d'aborder. Cependant, l'installation d'un serveur d'applications n'est pas une solution rapide à mettre en œuvre et que l'on peut tester facilement. JBoss propose un outil qui va nous être d'une grande utilité : JBoss intégré, ou *Embedded JBoss*. (Rendez-vous sur la page Web dédiée à l'ouvrage sur le site d'Eyrolles pour télécharger les codes source du livre, ainsi que JBoss intégré préconfiguré et prêt à l'emploi.)

La structure grosse maille de notre projet sera la suivante :

```
java-persistance
   +bootstrap
   +src
      +test
      +app
         +resources
         +java
   +lib
   +optionnal-lib
```

Après avoir abordé JBoss intégré, nous détaillerons chacun des éléments qui constituent cette structure.

JBoss intégré

Nous allons donc utiliser un outil JBoss élégant qui permet d'exécuter plusieurs composants de la plate-forme Entreprise dans un environnement léger et autonome. Cet outil est JBoss intégré, téléchargeable à l'adresse *http://wiki.jboss.org/wiki/Wiki.jsp?page=EmbeddedJBoss*.

JBoss intégré est un conteneur léger. Il est très facile à mettre en place et propose un nombre élevé de services que l'on peut initialiser « comme si » nous configurions un

serveur d'applications, le tout dans de simples tests unitaires depuis un environnement autonome.

Parmi ces composants, nous retrouvons :

- Datasource locale. Utiliser un pool de connexions arbitraire comme C3P0 (abordé au chapitre 10) requiert souvent l'acquisition des compétences spécifiques. Nous allons pouvoir en faire abstraction en exploitant une Datasource intégrée et simple à mettre en œuvre.
- JNDI (Java Naming and Directory Interface) local. Au lieu de stocker certains objets comme la Datasource dans des singletons ou de manière non standardisée, autant exploiter JNDI, qui est ici simple d'accès.
- Gestionnaire de transactions. Grâce à ce composant, nous allons pouvoir intégrer JTA à nos exemples de code et aller bien au-delà de la démarcation des transactions offerte par JDBC dans un environnement autonome.
- Intégration normalisée de Java Persistence *via* Hibernate. Cette intégration nous permettra de masquer Hibernate en écrivant nos entités en conformité totale avec le standard Java Persistence, mais aussi, à tout moment, d'attaquer l'implémentation Hibernate et ses spécificités.
- EJB 3.0. Au-delà des l'aspect persistance, nous aurons l'opportunité de d'invoquer des beans sessions avec et sans état, depuis nos tests unitaires.

D'autres services sont disponibles, mais ne seront pas étudiés dans ce livre. Il s'agit de JMS, des MDB et de la sécurité EJB.

JBoss intégré est facile à installer. Nous l'avons toutefois légèrement modifié pour vous permettre de mettre facilement à jour les bibliothèques Hibernate, notre implémentation de Java Persistence.

Mettre à jour l'implémentation de Java Persistence

Par défaut, JBoss intégré est distribué avec la liste des bibliothèques suivante :

```
Java-Persistance
java-persistance
   +bootstrap
   +src
   -lib
     hibernate-all.jar
     jboss-embedded-all.jar
     jboss-embedded-tomcat-bootstrap.jar
     thirdparty-all.jar
   -optionnal-lib
     junit.jar
     jboss-test.jar
```

Cette distribution regroupe toutes les classes nécessaires à Java Persistence par Hibernate. Ces classes sont packagées dans hibernate-all.jar. Si vous souhaitez

mettre Hibernate à jour, vous devez supprimer cette bibliothèque et la remplacer par l'ensemble constitué de ejb3-persistence.jar, hibernate-annotations.jar, hibernate-entitymanager.jar, hibernate3.jar, hibernate-commons-annotations.jar et hibernate-validator.jar. Vous pouvez télécharger sur le site Hibernate *(http://www.hibernate.org)* de nouvelles versions, le moyen le plus simple étant de télécharger la distribution d'Hibernate Entity Manager.

Une fois Hibernate mis à jour, la liste de bibliothèques est la suivante :

```
java-persistance
   +bootstrap
   +src
   -lib
     ejb3-persistence.jar
     hibernate-annotations.jar
     hibernate-commons-annotations.jar
     hibernate-entitymanager.jar
     hibernate-validator.jar
     hibernate3.jar
     jboss-embedded-all.jar
     jboss-embedded-tomcat-bootstrap.jar
     thirdparty-all.jar
   -optionnal-lib
     junit.jar
     jboss-test.jar
```

Assurez-vous de la compatibilité des composants Hibernate en suivant la matrice de compatibilité présentée sur la page *http://www.hibernate.org/6.html#A3.*

Il est probable qu'à compter de la version Hibernate 3.3, le packaging varie légèrement. Le site vous guidera dans le téléchargement des composants.

Comprendre les bibliothèques présentes dans la distribution Hibernate

Si vous souhaitez utiliser Hibernate en dehors de notre configuration, il est important de savoir à quoi correspondent les bibliothèques distribuées. En effet, lorsque vous téléchargez Hibernate, vous ne récupérez pas moins d'une quarantaine de bibliothèques (fichiers jar).

Le fichier _README.TXT, disponible dans le répertoire lib, vous permet d'y voir plus clair dans toutes ces bibliothèques. Le tableau 2.1 en donne une traduction résumée.

Nous traitons ici des bibliothèques distribuées avec le noyau Hibernate. En d'autres termes, nous utilisons Hibernate Entity Manager, qui est l'implémentation de Java Persistence. Ce projet est principalement dépendant du noyau Hibernate (Hibernate Core *via* Hibernate3.jar) et de Hibernate Annotations. Par conséquent, nous nous intéressons aux bibliothèques distribuées avec Hibernate Core, qui est le composant le plus important et représente le moteur de persistance.

Tableau 2.1 Bibliothèques présentes dans la distribution du noyau Hibernate

Catégorie	Bibliothèque	Fonction	Particularité
Indispensables à l'exécution	dom4j-xxx.jar	Parseur de configuration XML et de mapping	Exécution. Requis
	xml-apis.jar	API standard JAXP	Exécution. Un parser SAX est requis.
	commons-logging-xxx.jar	Commons Logging	Exécution. Requis
	jta.jar	API Standard JTA	Exécution. Requis pour les applications autonomes s'exécutant en dehors d'un serveur d'applications
	jdbc2_0-stdext.jar	API JDBC des extensions standards	Exécution. Requis pour les applications autonomes s'exécutant en dehors d'un serveur d'applications
	antlr-xxx.jar	ANTLR (ANother Tool for Language Recognition)	Exécution
	javaassist.jar	Générateur de bytecode javaassist	Exécution. Requis si vous choisissez le fournisseur de bytecode javaassist.
	asm.jar	Générateur de bytecode	Exécution. Requis si vous choisissez le fournisseur de bytecode cglib.
	asm-attrs	Générateur de bytecode	Exécution. Requis si vous choisissez le fournisseur de bytecode cglib.
	cglib-full-xxx.jar	Générateur de bytecode CGLIB	Exécution. Requis si vous choisissez le fournisseur de bytecode cglib.
	xerces-xxx.jar	Parser SAX	Exécution. Requis si votre JDK est supérieur à 1.4.
	commons-collections-xxx.jar	Collections Commons	Exécution. Requis
En relation avec le pool de connexions	c3p0-xxx.jar	Pool de connexions JDBC C3P0	Exécution. Optionnel
	proxool-xxx.jar	Pool de connexions JDBC Proxool	Exécution. Optionnel
En relation avec le cache (les jars liés à JBoss Cache son sujets à évolution)	ehcache-xxx.jar	Cache EHCache	Exécution. Optionnel. Requis si aucun autre fournisseur de cache n'est paramétré.
	jboss-cache.jar	Cache en clusters JBossCache	Exécution. Optionnel
	jboss-system.jar		Exécution. Optionnel. Requis par JBossCache

Tableau 2.1 Bibliothèques présentes dans la distribution du noyau Hibernate (suite)

Catégorie	Bibliothèque	Fonction	Particularité
	jboss-common.jar		Exécution. Optionnel. Requis par JBossCache
	jboss-jmx.jar		Exécution. Optionnel. Requis par JBossCache
	concurrent-xxx.jar		Exécution. Optionnel. Requis par JBossCache
	swarmcache-xxx.jar		Exécution. Optionnel
	jgroups-xxx.jar	Bibliothèque multicast JGroups	Exécution. Optionnel. Requis par les caches supportant la réplication
	oscache-xxx.jar	OSCache OpenSymphony	Exécution. Optionnel
Autres bibliothèques optionnelles	connector.jar (unknown)	API JCA standard	Exécution. Optionnel
	jaas.jar (unknown)	API JAAS standard	Exécution. Optionnel. Requis par JCA
	jacc-xxx.jar	Bibliothèque JACC	Exécution. Optionnel
	log4j-xxx.jar	Bibliothèque Log4j	Exécution. Optionnel
	jaxen-xxx.jar	Jaxen, moteur XPath Java universel	Exécution. Requis si vous souhaitez désérialiser. Configuration pour améliorer les performances au démarrage.
Indispensables pour construire Hibernate	versioncheck.jar	Vérificateur de version	Construction
	checkstyle-all.jar	Checkstyle	Construction
	junit-xxx.jar	Framework de test JUnit	Construction
	ant-launcher-xxx.jar	Launcher Ant	Construction
	ant-antlr-xxx.jar	Support ANTLR Ant	Construction
	syndiag2.jar	Générateur de l'image bnf de antlr	Construction
	cleanimports.jar (unknown)	Cleanimports	Construction
	ant-junit-xxx.jar	Support JUnit-Ant	Construction
	ant-swing-xxx.jar	Support Swing-Ant	Construction
	ant-xxx.jar	Core Ant	Construction

Pour vous permettre d'y voir plus clair dans la mise en place de vos projets de développement, que vous souhaitiez modifier Hibernate puis le recompiler, l'utiliser dans sa version la plus légère ou brancher un cache et un pool de connexions, la figure 2.1 donne une synthèse visuelle de ces bibliothèques.

À ces bibliothèques s'ajoutent hibernate3.jar ainsi que le pilote JDBC indispensable au fonctionnement d'Hibernate. Plus le pilote JDBC est de bonne qualité, meilleures sont les performances, Hibernate ne corrigeant pas les potentiels bogues du pilote.

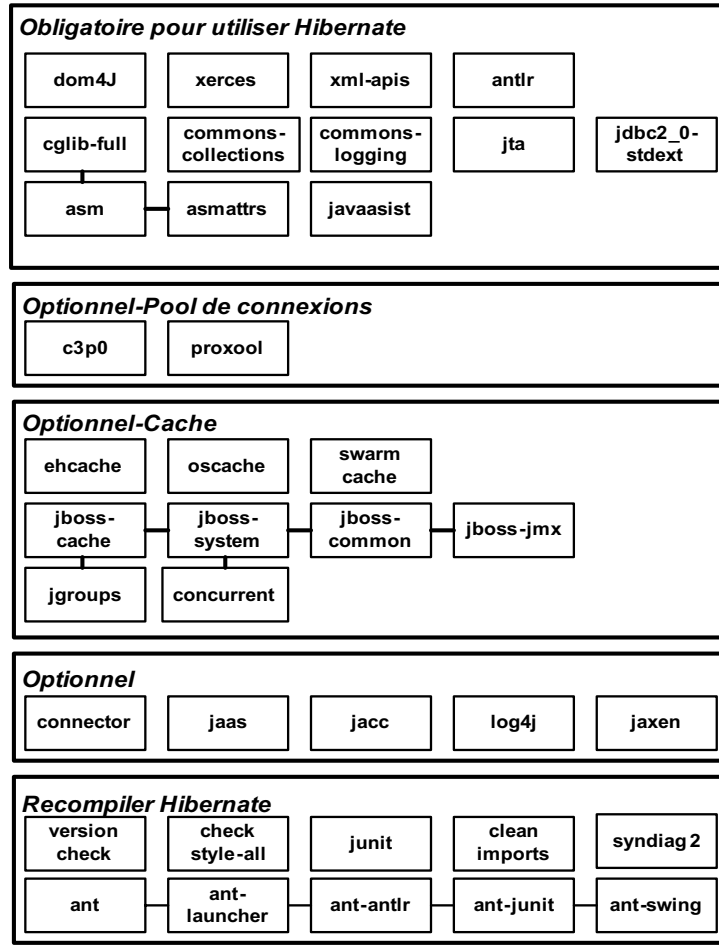


Figure 2-1
Bibliothèques distribuées avec le noyau Hibernate

Configurations

Une unité de persistance définit un ensemble de classes exploitées par une application et qui sont liées à une même base de données. Une application peut utiliser plusieurs unités de persistance. Les unités de persistance sont définies dans un fichier nommé persistence.xml, qui doit se trouver dans le répertoire /META-INF. L'archive ou le répertoire contenant le répertoire /META-INF est nommée « racine de l'unité de persistance ».

Dans un environnement entreprise Java (JEE) cette racine peut être :

- un fichier EJB-jar ;
- le répertoire WEB-INF/classes d'un fichier war ;
- un fichier jar à la racine d'un ear ;

- un fichier jar dans le répertoire des bibliothèques d'un ear ;
- un fichier jar d'une application cliente.

Une unité de persistance est :

- définie dans META-INF/persistence.xml ;
- responsable de la gestion d'un ensemble d'entités (classes persistantes) ;
- liée à une même base de données.

La définition de la correspondance entre les entités et les tables est faite *via* les métadonnées.

Voyons désormais les différentes étapes des configurations.

Configurations globales de JBoss intégré

Avant de détailler les configurations importantes spécifiques à notre application, rappelons que JBoss intégré propose une multitude de services.

Ces services sont paramétrables *via* les fichiers qui se trouvent dans le répertoire / bootstrap :

```
Java-Persistance
   -bootstrap
     jndi.properties
     log4j.xml
     -conf
       bootstrap-beans.xml
       jboss-service.xml
       jbossjta-properties.xml
       login-config.xml
       -props
         messaging-roles.properties
         messaging-users.properties
     -data
       …
     -deploy
       ejb3-interceptors-aop.xml
       hsqldb-ds.xml
       jboss-local-jdbc.rar
       jboss-xa-jdbc.rar
       jms-ra.rar
       messaging
       remoting-service.xml
       -messaging
         connection-factories-service.xml
         destinations-service.xml
         hsqldb-persistence-service.xml
         jms-ds.xml
         legacy-service.xml
         messaging-service.xml
         remoting-service.xml
```

```
      -deployers
          aspect-deployer-beans.xml
          ejb3-deployers-beans.xml
          jca-deployers-beans.xml
   +resources
   +lib
   +src
```

Comme vous le voyez, un certain nombre de fichiers de configuration sont nécessaires au paramétrage de JBoss intégré. La bonne nouvelle est que nous n'aurons à modifier aucun de ces fichiers, si ce n'est le fichier log4j.xml pour affiner le paramétrage des traces.

Définition d'une source de données

Nous allons configurer une source de données spécifique à notre application, qui se trouve dans notre répertoire src/app/resources/myDS-ds.xml :

```
java-persistance
   +bootstrap
   -src
      -test
      -app
         -resources
           myDS-ds.xml
           eyrolles-persistence.xml
         +java
   +lib
   +optionnal-lib
```

Son contenu est le suivant :

```
< ?xml version="1.0" encoding="UTF-8"?>
<datasources>
   <local-tx-datasource>
      <jndi-name>myDS</jndi-name>
      <connection-url>jdbc:hsqldb:.</connection-url>
      <driver-class>org.hsqldb.jdbcDriver</driver-class>
      <user-name>sa</user-name>
      <min-pool-size>1</min-pool-size>
      <max-pool-size>10</max-pool-size>
      <idle-timeout-minutes>0</idle-timeout-minutes>
   </local-tx-datasource>
</datasources>
```

Comme nous utilisons HSQLDB comme base de données *(http://hsqldb.org/),* nous avons pris soin de placer le pilote jdbc (hsqldb.jar) dans le répertoire lib du projet java-persistance.

Dans le cadre de l'application test, selon votre instance de base de données, vous devrez modifier les propriétés suivantes :

- `DriverClass` : nom du pilote jdbc.

- `ConnectionURL` : URL de connexion à votre base de données.
- `userName` : nom de connexion à votre base de données.
- `password` : mot de passe pour vous connecter à la base de données ; dans notre cas, cette propriété n'est pas déclarée car égale à la chaîne de caractères vide.

Voyons désormais comment établir le lien entre le moteur de persistance et notre source de données.

Paramétrer une unité de persistance

L'autre fichier spécifique à notre application est semi-normalisé. Il doit être nommé persistence.xml et se trouver dans /META-INF.

La configuration d'une unité de persistance comprend deux parties.

Configuration normalisée

La première partie du fichier META-INF/persistence.xml est normalisée et contient :

- le nom de l'unité de persistance <persistence-unit name="…">.
- le nom JNDI de la source de données et son type (JTA ou non-JTA) <jta-data-source/> ou <non-jta-data-source/>.
- un élément informatif <description/>.
- le nom du fournisseur de l'implémentation <provider/>.
- une série de quatre éléments qui permettent d'affiner la prise en compte des métadonnées : <mapping-file/>, <class/>, <jar-file/> et <exclude-unlisted-class/>.

Les métadonnées sont abordées en détail au chapitre 3. Ce sont des informations permettant le mapping entre les entités et la base de données. Les métadonnées se présentent sous diverses formes. La plus performante en termes de développement est les annotations, que vous insérez directement dans le code de vos entités. Ce livre ne traite que des annotations.

Dans l'hypothèse où vous ne pouvez ou ne souhaitez pas exploiter les annotations, vous devez « externaliser » ces informations dans des descripteurs de déploiement XML. Il en existe de deux sortes. La première est normalisée et requiert la création d'un fichier META-INF/orm.xml, au format verbeux et beaucoup moins intuitif que les annotations ; il est détaillé dans la spécification. La seconde est propriétaire à Hibernate : il s'agit des fichiers de déploiement hbm.xml, dont le format est disponible dans la documentation officielle Hibernate. (La page Web dédiée au livre du site Web d'Eyrolles propose un référentiel des métadonnées au format spécifique Hibernate.hbm.xml.)

Dans un environnement EE, l'unité de persistance détecte toutes les métadonnées présentes dans l'application déployée, qu'elles se présentent sous la forme d'annotations ou de fichiers XML.

Dans un environnement SE, cette autodétection n'est pas requise, et il vous faudra renseigner les nœuds <mapping-file/>, <class/> et <jar-file/>, comme le montre l'exemple suivant :

```
<persistence … >
   <persistence-unit name="manager1" transaction-type="JTA">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <mapping-file>ormap.xml</mapping-file>
      <jar-file>MyApp.jar</jar-file>
      <class>org.acme.Employee</class>
      <class>org.acme.Person</class>
      <class>org.acme.Address</class>
      …
   </persistence-unit>
</persistence>
```

Cependant, même en environnement SE, Hibernate est assez puissant pour effectuer l'autodétection. Ces options de configuration ne devraient donc pas vous être utiles.

Vous pouvez désactiver l'autodétection et passer en mode manuel en activant le nœud <exclude-unlisted-class/>.

Métadonnées

Les métadonnées permettent de définir la correspondance entre les entités et la base de données relationnelle.

Définissables sous trois formes, les annotations intégrées au code, le fichier de déploiement normalisé META-INF/orm.xml ou les fichiers de mapping spécifiques Hibernate.hbm.xml, elles sont détectées automatiquement dans un environnement EE et à définir *via* <mapping-file/>, <class/> et <jar-file/> dans un environnement SE (inutile si vous choisissez Hibernate comme fournisseur de l'implémentation).

Les exemples qui illustrent ce livre tirent profit de la détection automatique des métadonnées. Étant donné le nombre important d'entités composant les divers exemples, nous choisirons, exemple par exemple, quelles entités doivent être déployées. Nous détaillons davantage ce mécanisme ultérieurement.

Configuration spécifique

La seconde partie du fichier META-INF/persistence.xml est spécifique au fournisseur de l'implémentation et passe par le nœud XML <properties/>, qui contient les propriétés propres à l'implémentation. Dans notre exemple, nous avons les propriétés hibernate.dialect et hibernate.hbm2ddl.auto.

Les tableaux 2.2 à 2.6 donnent la liste des paramètres disponibles pour l'implémentation Hibernate**.**

Le tableau 2.2 récapitule les paramètres JDBC à mettre en place. Utilisez-les si vous ne disposez pas d'une datasource. Dans notre cas, il est inutile puisque nous exploitons une source de données mise à disposition par JBoss intégré *via* le registre JNDI. Dans le cas

d'une utilisation « out of the box », ce paramétrage nécessite de choisir un pool de connexions. Hibernate est livré avec C3P0 et Proxool. Un exemple d'utilisation de pool de connexions est traité au chapitre 10.

Tableau 2.2 Paramétrage JDBC

Paramètre	Rôle
connection.driver_class	Classe du pilote JDBC
connection.url	URL JDBC
connection.username	Utilisateur de la base de données
connection.password	Mot de passe de l'utilisateur spécifié
connection.pool_size	Nombre maximal de connexions poolées, si utilisation du pool Hibernate (déconseillé en production)

Le tableau 2.3 reprend les paramètres relatifs à l'utilisation d'une datasource. Si vous utilisez un serveur d'applications, préférez la solution datasource à un simple pool de connexions. Une fois encore, notre environnement intégré nous décharge de ce paramétrage.

Tableau 2.3 Paramétrage de la datasource

Paramètre	Rôle
hibernate.connection.data-source	Nom JNDI de la datasource
hibernate.jndi.url	URL du fournisseur JNDI (optionnel)
hibernate.jndi.class	Classe de la InitialContextFactory JNDI
hibernate.connection.username	Utilisateur de la base de données
hibernate.connection.password	Mot de passe de l'utilisateur spécifié

Le tableau 2.4 donne la liste des bases de données relationnelles supportées et le dialecte à paramétrer pour adapter la génération du code SQL aux spécificités syntaxiques de la base de données. Le dialecte est crucial.

Tableau 2.4 Bases de données et dialectes supportés

SGBD	Dialecte
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Oracle (toutes versions)	org.hibernate.dialect.OracleDialect (à dater d'Hibernate 3.3, ce dialecte pourrait être déprécié ; lisez les releases notes).

Tableau 2.4 Bases de données et dialectes supportés (suite)

SGBD	Dialecte
Oracle 9/10g	org.hibernate.dialect.Oracle9Dialect (à dater d'Hibernate 3.3, ce dialecte pourrait être déprécié ; lisez les releases notes).
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
Firebird	org.hibernate.dialect.FirebirdDialect

Le tableau 2.5 récapitule les différents gestionnaires de transactions disponibles en fonction du serveur d'applications utilisé, inutile dans notre cas.

Tableau 2.5 Gestionnaires de transaction

Serveur d'applications	Gestionnaire de transaction
JBoss	org.hibernate.transaction.JBossTransactionManagerLookup
WebLogic	org.hibernate.transaction.WeblogicTransactionManagerLookup
WebSphere	org.hibernate.transaction.WebSphereTransactionManagerLookup
Orion	org.hibernate.transaction.OrionTransactionManagerLookup
Resin	org.hibernate.transaction.ResinTransactionManagerLookup
JOTM	org.hibernate.transaction.JOTMTransactionManagerLookup
JOnAS	org.hibernate.transaction.JOnASTransactionManagerLookup
JRun4	org.hibernate.transaction.JRun4TransactionManagerLookup
Borland ES	org.hibernate.transaction.BESTransactionManagerLookup

Le tableau 2.6 recense l'ensemble des paramètres optionnels. Cette liste évoluant régulièrement, il est conseillé de se référer au guide de référence.

Tableau 2.6 Principaux paramètres optionnels

Paramètre	Rôle
hibernate.dialect	Classe d'un dialecte Hibernate
hibernate.default_schema	Qualifie (dans la génération SQL) les noms des tables non qualifiées avec le schema/tablespace spécifié.
hibernate.default_catalog	Qualifie (dans la génération SQL) les noms des tables avec le catalogue spécifié.
hibernate.session_factory_name	La SessionFactory est automatiquement liée à ce nom dans JNDI après sa création.
hibernate.max_fetch_depth	Active une profondeur maximale de chargement par outer-join pour les associations simples (one-to-one, many-to-one). 0 désactive le chargement par outer-join.
hibernate.fetch_size	Une valeur différente de 0 détermine la taille de chargement JDBC (appelle Statement.setFetchSize()).
hibernate.batch_size	Une valeur différente de 0 active l'utilisation des updates batch de JDBC2 par Hibernate. Il est recommandé de positionner cette valeur entre 3 et 30.
hibernate.batch_versioned_data	Définissez ce paramètre à true si votre pilote JDBC retourne le nombre correct d'enregistrements à l'exécution de executeBatch().
hibernate.use_scrollable_resultset	Active l'utilisation des *scrollable resultsets* de JDBC2. Ce paramètre n'est nécessaire que si vous gérez vous-même les connexions JDBC. Dans le cas contraire, Hibernate utilise les métadonnées de la connexion.
hibernate.jdbc.use_streams_for_binary	Utilise des flux lorsque vous écrivez/lisez des types binary ou serializable vers et à partir de JDBC (propriété de niveau système).
hibernate.jdbc.use_get_generated_keys	Active l'utilisation de PreparedStatement.getGeneratedKeys() de JDBC3 pour récupérer nativement les clés générées après insertion. Nécessite un driver JDBC3+. Mettez-le à false si votre driver rencontre des problèmes avec les générateurs d'identifiants Hibernate. Par défaut, essaie de déterminer les possibilités du driver en utilisant les métadonnées de connexion.
hibernate.cglib.use_reflection_optimizer	Active l'utilisation de CGLIB à la place de la réflexion à l'exécution (propriété de niveau système ; la valeur par défaut est d'utiliser CGLIB lorsque c'est possible). La réflexion est parfois utile en cas de problème.
hibernate.jndi.<propertyName>	Passe la propriété propertyName au JNDI InitialContextFactory.
hibernate.connection.<propertyName>	Passe la propriété JDBC propertyName au DriverManager.getConnection().
hibernate.connection.isolation	Positionne le niveau de transaction JDBC. Référez-vous à java.sql.Connection pour le détail des valeurs, mais sachez que toutes les bases de données ne supportent pas tous les niveaux d'isolation.
hibernate.connection.provider_class	Nom de classe d'un ConnectionProvider spécifique
hibernate.hibernate.cache.provider_class	Nom de classe d'un CacheProvider spécifique
hibernate.cache.use_minimal_puts	Optimise le cache de second niveau en minimisant les écritures, mais au prix de davantage de lectures (utile pour les caches en cluster).

Tableau 2.6 Principaux paramètres optionnels (suite)

Paramètre	Rôle
hibernate.cache.use_query_cache	Active le cache de requête. Les requêtes individuelles doivent tout de même être déclarées comme susceptibles d'être mises en cache.
hibernate.cache.region_prefix	Préfixe à utiliser pour le nom des régions du cache de second niveau
hibernate.transaction.factory_class	Nom de classe d'une TransactionFactory qui sera utilisé par l'API Transaction d'Hibernate (la valeur par défaut est JDBCTransactionFactory).
jta.UserTransaction	Nom JNDI utilisé par la JTATransactionFactory pour obtenir la UserTransaction JTA du serveur d'applications
hibernate.transaction.manager_lookup_class	Nom de la classe du TransactionManagerLookup. Requis lorsque le cache de niveau JVM est activé dans un environnement JTA
hibernate.query.substitutions	Lien entre les tokens de requêtes Hibernate et les tokens SQL. Les tokens peuvent être des fonctions ou des noms littéraux. Exemples : hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC.
hibernate.show_sql	Écrit les ordres SQL dans la console.
hibernate.hbm2ddl.auto	Exporte automatiquement le schéma DDL vers la base de données lorsque la SessionFactory est créée. La valeur create-drop permet de supprimer le schéma de base de données lorsque la SessionFactory est explicitement fermée.
hibernate.transaction.manager_lookup_class	Nom de la classe du TransactionManagerLookup. Requis lorsque le cache de niveau JVM est activé dans un environnement JTA.

Les paramètres de configuration spécifiques à Hibernate sont désormais définis. Vous ferez momentanément abstraction des annotations, qui permettent de mettre en correspondance vos classes Java et votre modèle relationnel. Ces annotations sont abordées à la section suivante.

Fichier persistence.xml de notre application exemple

Notre application utilisera le fichier de configuration suivant :

```
< ?xml version="1.0" encoding="UTF-8"?>
<persistence>
   <persistence-unit name="eyrollesEntityManager">
      <jta-data-source>java:/myDS</jta-data-source>
      <properties>
         <property name="hibernate.hbm2ddl.auto"
           value="create-drop"/>
         <property name="hibernate.dialect"
           value="org.hibernate.dialect.HSQLDialect"/>
         <property name="jboss.entity.manager.jndi.name"
           value="java:/EntityManagers/eyrollesEntityManager"/>
      </properties>
   </persistence-unit>
</persistence>
```

Le nom de l'unité de persistance est requis et sera utilisé pour l'injection dans les beans session que nous verrons plus tard.

Nous exploitons une source de données JTA localisable dans le registre JNDI sous le nom java:/myDS.

Nous utiliserons ensuite un outil intégré à Hibernate, qui permet de générer le schéma de base de données depuis l'analyse des métadonnées. Pour cela, nous paramétrons hibernate.hbm2ddl.auto à la valeur create-drop.

SchemaExport

SchemaExport est une pièce maîtresse des exemples qui illustrent ce livre. Lorsqu'une unité de persistance est initialisée, les métadonnées sont analysées. À partir de ce moment, toutes les informations concernant la base de données sont disponibles et un schéma peut être généré.

SchemaExport se charge de créer, à la volée, un tel schéma avant l'exécution de votre code.

Nous revenons en détail sur SchemaExport au chapitre 9. Pour le moment, sachez qu'il travaille en tâche de fond pour, à chaque exemple, créer et injecter le schéma nécessaire.

Enfin, nous définissons que le gestionnaire d'entités courant doit être placé dans le registre JNDI sous le nom java:/EntityManagers/eyrollesEntityManager.

Exploiter un fichier Hibernate.cfg.xml

Il est possible d'exploiter un fichier de configuration globale Hibernate en paramétrant :

```
<property name="hibernate.ejb.cfgfile" value="/hibernate.cfg.xml"/>
```

Notre unité de persistance étant désormais configurée, nous allons pouvoir l'exploiter dans notre code.

Un mot sur JUnit

Comme nous l'avons indiqué précédemment, le choix du socle de développement JBoss intégré est le meilleur rapport entre la fidélité à la spécification, la simplicité de mise en œuvre, la légèreté de l'architecture et la possibilité d'appréhender un maximum de fonctionnalités, que ce soit en terme de mapping, *via* les fonctionnalités Hibernate, ou d'autres composants de la plate-forme entreprise, comme JTA et les beans session.

Nous allons utiliser JUnit, qui, de manière transparente, initialisera JBoss intégré, lequel à son tour initialisera tous les services dont nous avons besoin. À votre charge de passer dynamiquement les classes, principalement les entités et beans session à déployer « virtuellement » par la suite. Nos classes de test hériteront d'une classe fournie par JBoss intégré : org.jboss.embedded.junit.BaseTestCase.

Étudiez bien la méthode `deploy()`, qui permet de spécifier quelles classes doivent être déployées au lancement du test :

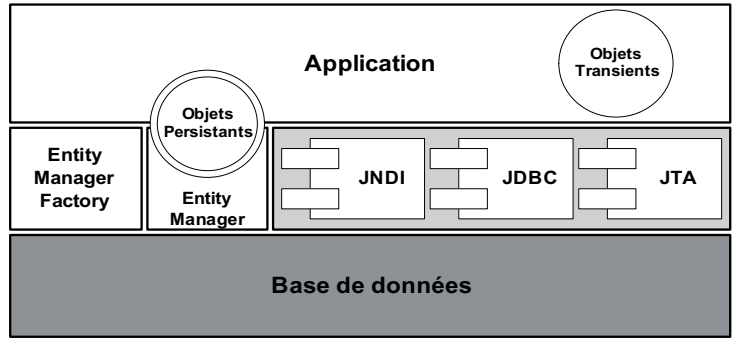
```
public static void deploy(){
  jar = AssembledContextFactory
    .getInstance().create("ejbTestCase.jar");

  //déploiement des entités
  jar.addClass(Customer.class);
  //déploiement des beans session
  jar.addClass(CustomerDAOBean.class);
  jar.addClass(CustomerDAOLocal.class);
  jar.addClass(CustomerDAORemote.class);
  jar.addResource("myDS-ds.xml");
  jar.mkdir("META-INF")
    .addResource("eyrolles-persistence.xml", "persistence.xml");

  try{
    Bootstrap.getInstance().deploy(jar);
  }
  catch (DeploymentException e) {
    throw new RuntimeException("Unable to deploy", e);
  }
}
```

Vous pouvez dès à présent télécharger les projets de test depuis le site Eyrolles. Importez java-persistance et java-persistance-se dans votre IDE, et configurez-les. Dans le projet java-persistance, exécutez le test ch2Tests.EjbTestCase.testEntityManager() depuis votre IDE pour vérifier que votre installation fonctionne correctement.

L'architecture autour de Java Persistence

Les principaux composants indispensables pour exploiter Java Persistence sont illustrés à la figure 2.2.

Figure 2-2
Composants de l'architecture Java Persistence



Cette architecture peut être résumée de la façon suivante : votre application dispose d'entités, dont la persistance est gérée cas d'utilisation par cas d'utilisation par un

gestionnaire d'entités. Rappelons qu'un objet est dit persistant lorsque sa durée de vie est longue, à l'inverse d'un objet transient, qui est temporaire.

Entité

Une entité est une classe persistante. Nous parlons d'entité aussi bien pour désigner la classe persistante que ses instances. Un objet qui n'est pas persistant est dit transient.

Un gestionnaire d'entités (EntityManager) s'obtient *via* une EntityManagerFactory ; l'EntityManagerFactory est construite à partir d'un objet EJB3Configuration et contient les informations de configuration globale ainsi que les informations contenues dans les annotations.

Lien entre Java Persistence et Hibernate

Java Persistence est une spécification, une définition d'interfaces. Son implémentation est Hibernate. Aussi est-il important d'avoir en tête les équivalences entre les termes employés lorsqu'on traite de Java Persistence et ceux correspondant à Hibernate :

- EntityManager correspond à la Session Hibernate. Lorsque vous manipulez un gestionnaire d'entités, sachez qu'une session Hibernate lui est indispensable, même si celle-ci est transparente à vos yeux.
- EntityManagerFactory correspond à la SessionFactory Hibernate.
- EntityTransaction correspond à la Transaction Hibernate.
- Ejb3Configuration correspond à la Configuration Hibernate.

Basculer en mode Hibernate natif

Vous pouvez très facilement basculer dans les API natives Hibernate depuis le gestionnaire d'entités grâce à la méthode standardisée `Session sessionHibernate = (Session)entityManager.getDelegate()`

Nous aurons l'occasion de l'exploiter dans diverses sections du livre.

Un gestionnaire d'entités effectue des opérations, dont la plupart ont un lien direct avec la base de données. Ces opérations se déroulent au sein d'une *transaction*.

Grâce à notre environnement proche du serveur d'applications, mais ultraléger, obtenir un gestionnaire d'entités est d'une grande simplicité. Vous récupérez le gestionnaire depuis le registre JNDI (dans notre environnement sous l'alias java:/EntityManagers/eyrollesEntityManager). Par contre, pour agir sur le gestionnaire d'entités, une transaction doit être entamée, sinon une exception est soulevée. En effet, le gestionnaire d'entités est sémantiquement lié à la transaction en cours.

L'EntityManagerFactory est coûteuse à instancier puisqu'elle requiert, entre autres, l'analyse des métadonnées. Sans conteneur, il est donc judicieux de la stocker dans un singleton (une seule instance pour l'application) en utilisant des variables statiques. Cette factory est *threadsafe,* c'est-à-dire qu'elle peut être attaquée par plusieurs traitements en parallèle. Là encore, notre environnement nous simplifiera la tâche en masquant la construction de l'EntityManagerFactory.

Le gestionnaire d'entités, lui, n'est pas threadsafe, mais il est peu coûteux à instancier. Il est en outre possible de le récupérer sans impacter les performances de votre application.

L'exemple de code suivant met en œuvre très facilement, grâce à JBoss intégré, les trois étapes décrites ci-dessus :

```
// récupération du gestionnaire d'entités
EntityManager em =
  (EntityManager) new InitialContext()
     .lookup("java:/EntityManagers/eyrollesEntityManager");

// récupération d'une transaction
TransactionManager tm =
  (TransactionManager) new InitialContext()
     .lookup("java:/TransactionManager");

// début de la transaction
tm.begin();

// instanciation d'une entité
Customer cust = new Customer();
cust.setName("Bill");

// rendre l'entité persistante
em.persist(cust);
tm.commit() ;
```

Nous récupérons le gestionnaire d'entités de l'utilisateur courant (pour simplifier, du thread courant) depuis JNDI. De la même manière, nous récupérons la transaction (JTA) de l'utilisateur courant, toujours depuis JNDI. Nous démarrons la transaction *via* `tm.begin()`. À ce stade, les deux objets, gestionnaire d'entités et transaction, semblent indépendants. En fait, c'est le conteneur JBoss intégré qui effectue le lien entre les deux. Par souci de lisibilité, ce canevas exemple de code que nous utiliserons jusqu'au chapitre 7 est incomplet puisqu'il ne traite pas les potentielles exceptions. Référez-vous au chapitre 7 pour une explication détaillée sur ce point.

À partir de ce moment, nous pouvons manipuler le gestionnaire d'entités. Tout le travail entrepris *via* le gestionnaire d'entités sera transmis et définitivement validé lors de l'appel à `tm.commit()`.

Les logs suivants s'affichent lors de l'initialisation de Java Persistence :

```
DEBUG [Ejb3Deployment] Bound ejb3 container
jboss.j2ee:jar=ejbTestCase.jar,name=CustomerDAOBean,service=EJB3

INFO  [PersistenceUnitDeployment] Starting persistence unit
persistence.units:jar=ejbTestCase.jar,unitName=eyrollesEntityManager

DEBUG [PersistenceUnitDeployment] Found persistence.xml file in EJB3 jar

INFO  [Version] Hibernate Annotations 3.3.0.GA
```

```
INFO  [Environment] Hibernate 3.2.4.sp1
INFO  [Environment] hibernate.properties not found
INFO  [Environment] Bytecode provider name : javassist
INFO  [Environment] using JDK 1.4 java.sql.Timestamp handling
INFO  [Version] Hibernate EntityManager 3.3.1.GA
INFO  [Ejb3Configuration] Processing PersistenceUnitInfo [
name: eyrollesEntityManager
...]
INFO  [Ejb3Configuration] found EJB3 Entity bean:
org.jboss.embedded.tutorial.junit.beans.Customer
INFO  [AnnotationBinder] Binding entity from annotated class:
org.jboss.embedded.tutorial.junit.beans.Customer
INFO  [EntityBinder] Bind entity
org.jboss.embedded.tutorial.junit.beans.Customer on table Customer
…
INFO  [Dialect] Using dialect: org.hibernate.dialect.HSQLDialect
INFO  [TransactionFactoryFactory] Transaction strategy:
org.hibernate.ejb.transaction.JoinableCMTTransactionFactory
INFO  [TransactionManagerLookupFactory] instantiating
TransactionManagerLookup:
org.hibernate.transaction.JBossTransactionManagerLookup
INFO  [TransactionManagerLookupFactory] instantiated TransactionManagerLookup
INFO  [SettingsFactory] Automatic flush during beforeCompletion(): disabled
INFO  [SettingsFactory] Automatic session close at end of transaction: disabled
…
INFO  [SettingsFactory] Connection release mode: auto
INFO  [SettingsFactory] Default batch fetch size: 1
INFO  [SettingsFactory] Generate SQL with comments: disabled
INFO  [SettingsFactory] JPA-QL strict compliance: enabled
INFO  [SettingsFactory] Second-level cache: enabled
INFO  [SettingsFactory] Query cache: disabled
…
INFO  [SessionFactoryObjectFactory] Factory name:
persistence.units:jar=ejbTestCase.jar,unitName=eyrollesEntityManager
INFO  [NamingHelper] JNDI InitialContext
properties:{java.naming.factory.initial=org.jnp.interfaces.NamingContextFacto
ry, java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces}
INFO  [SessionFactoryObjectFactory] Bound factory to JNDI name:
persistence.units:jar=ejbTestCase.jar,unitName=eyrollesEntityManager
```

```
WARN  [SessionFactoryObjectFactory] InitialContext did not implement
EventContext
INFO  [SchemaExport] Running hbm2ddl schema export
INFO  [SchemaExport] exporting generated schema to database
INFO  [SchemaExport] schema export complete
```

Ces logs vous permettent d'appréhender tous les détails de configuration pris en compte par Hibernate d'une manière relativement lisible ; cela va de la détection de la définition d'une unité de persistance à la création du schéma de la base de données en passant par l'analyse des entités.

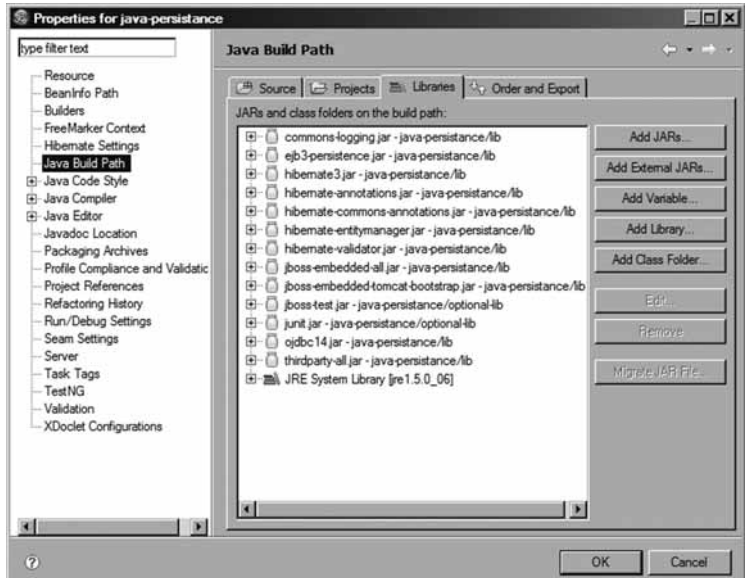
Comme vous l'avez vu, obtenir un gestionnaire d'entités est chose facile. Nous verrons que cela peut être encore simplifié grâce à l'injection du gestionnaire d'entités dans les beans session EJB3. Il existe divers moyens d'obtenir un gestionnaire d'entités selon votre environnement. Ces méthodes sont abordées en détail au chapitre 7.

En résumé

Pour vos premiers pas avec Java Persistence, vous disposez des prérequis pour monter un projet, par exemple, sous Eclipse, en suivant la structure décrite tout au long de cette partie ainsi que les bibliothèques à prendre en compte illustrées à la figure 2.3. Il vous suffit ensuite d'écrire le fichier META-INF/persistence.xml ainsi que d'annoter vos entités.

Pour le moment, c'est le registre JNDI qui vous fournira le gestionnaire d'entités avec lequel vous interagirez de manière transparente avec la base de données.

Figure 2-3
Bibliothèques à prendre en compte



Abordons désormais le cœur de notre problématique : les entités.

Les entités

Java est un langage orienté objet. L'un des objectifs de l'approche orientée objet est de découper une problématique globale en un maximum de petits composants, chacun de ces composants ayant la charge de participer à la résolution d'une sous-partie de la problématique globale. Ces composants sont les objets. Leurs caractéristiques et les services qu'ils doivent rendre sont décrits dans des classes.

On appelle modèle de classes métier un modèle qui définit les problématiques réelles rencontrées par une application. La plupart de ces classes donnent naissance à des instances persistantes : ce sont les entités.

Les avantages attendus d'une conception orientée objet sont, entre autres, la maintenance facilitée, puisque le code est factorisé, et donc localisable en un point unique (cas idéal), mais aussi la réutilisabilité, puisqu'une micro-problématique résolue par un composant n'a plus besoin d'être réinventée par la suite. La réutilisabilité demande un investissement constant dans vos projets.

> **Pour en savoir plus**
>
> Vous trouverez un article intéressant sur la réutilisation dans l'analyse faite par Jeffrey S. Poulin sur la page *http://home.stny.rr.com/jeffreypoulin/Papers/WISR95/wisr95.html.*

Le manque de solution de persistance totalement transparente ou efficace ajouté à la primauté prise par les bases de données relationnelles sur les bases de données objet ont fait de la persistance des données l'un des problèmes, si ce n'est le problème, majeur des développements d'applications informatiques.

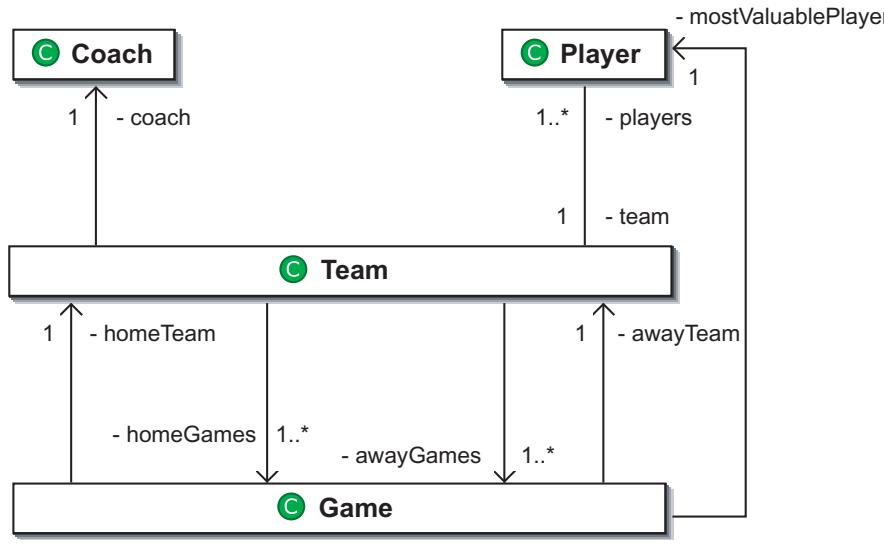
Les classes composant le modèle de classes métier d'une application informatique ne doivent pas consister en une simple définition de propriétés aboutissant dans une base de données relationnelle. Il convient plutôt d'inverser cette définition de la façon suivante : les classes qui composent votre application doivent rendre un service, lequel s'appuie sur des propriétés, certaines d'entre elles devant durer dans le temps. En ce sens, la base de données est un moyen de faire durer des informations dans le temps et n'est qu'un élément de stockage, même si cet élément est critique.

Une classe métier ne se contente donc pas de décrire les données potentiellement contenues par ses instances. Les entités sont les classes dont les instances doivent durer dans le temps. Ce sont celles qui sont prises en compte par le gestionnaire d'entités. Les éléments qui composent ces entités sont décrits dans les métadonnées.

Exemple de diagramme de classes

Après ce rappel sur les aspects persistants et métier d'une classe, prenons un exemple ludique. Le diagramme de classes illustré à la figure 2.4 illustre un sous-ensemble de l'application de gestion d'équipes de sports introduite au chapitre 1.

Figure 2-4
Diagramme de classes exemple



Le plus important dans ce diagramme réside dans les liens entre les classes, la navigabilité et les rôles qui vont donner naissance à des propriétés. Dans le monde relationnel, nous parlerions de tables contenant des colonnes, potentiellement liées entre elles. Ici, notre approche est résolument orientée objet.

Détaillons la classe Team, et découpons-la en plusieurs parties :

```
package chapitre2ModeleExemple;
// imports nécessaires

/**
 * Une instance de cette classe représente une Team.
 * Des players et des games sont associés à cette team.
 * Son cycle de vie est indépendant de celui des objets associés
 * @author Anthony Patricio <anthony@hibernate.org>
 */
public class Team implements Serializable{
  private Long id;←❶
  private String name;←❷
  private int nbWon;←❷
  private int nbLost;←❷
  private int nbPlayed;←❷
  private Coach coach;←❸
  private Set<Player> players = new HashSet<Player>();←❹
  // nous verrons plus tard comment choisir la collection
  // private List players = new ArrayList
  private Map<Date,Game> homeGames = new HashMap<Date,Game>();←❹
  private Map<Date,Game> awayGames = new HashMap<Date,Game>();←❹
  private transient int nbNull;←❺
  private transient Map<Date,Game> games
       = new HashMap<Date,Game> ();←❺
```

```
    private transient Set<Game> wonGames = new HashSet<Game> ();←❺
    private transient Set<Game> lostGames = new HashSet<Game> ();←❺

  /**
   * Contructeur par défaut←❻
   */
   public Team() {}

  // méthodes métier←❼
  // getters & setters←❽
  // equals & hashcode←❾
```

Cette classe est une entité. Cela veut dire que ses instances vont être stockées dans un entrepôt, ou datastore. Concrètement, il s'agit de la base de données relationnelle. À n'importe quel moment, une application est susceptible de récupérer ces instances, qui sont aussi dites persistantes, à partir de son identité (repère ❶) dans le datastore. Cette identité peut être reprise comme propriété de la classe.

La classe décrit ensuite toutes les propriétés dites simples (repère ❷) puis les associations vers un objet (repère ❸) et fait de même avec les collections (repère ❹). Elle peut contenir des propriétés calculées (repère ❺). Le repère ❻ indique le constructeur par défaut, le repère ❼ les méthodes métier et le repère ❽ les accesseurs (getters et setters).

Les méthodes `equals` et `hashcode` (repère ❾) permettent d'implémenter les règles d'égalité de l'objet. Il s'agit de spécifier les conditions permettant d'affirmer que deux instances doivent être considérées comme identiques ou non.

Chacune de ces notions a son importance, comme nous allons le voir dans les sections suivantes.

L'unicité métier

La notion d'unicité est commune aux mondes objet et relationnel. Dans le monde relationnel, elle se retrouve dans deux contraintes d'intégrité. Pour rappel, une contrainte d'intégrité vise à garantir la cohérence des données.

Pour assurer l'unicité d'un enregistrement dans une table, ou tuple, la première contrainte qui vient à l'esprit est la clé primaire. . Une clé primaire *(primary key)* est un ensemble de colonnes dont la combinaison forme une valeur unique. La contrainte d'unicité *(unicity constraint)* possède exactement la même signification.

Dans notre application exemple, nous pourrions dire pour la table `TEAM` qu'une colonne `NAME`, contenant le nom de la *team,* serait bonne candidate à être une clé primaire. D'après le diagramme de classes de la figure 2.4, il est souhaitable que les tables `COACH` et `PLAYER` possèdent une référence vers la table `TEAM` du fait des associations. Concrètement, il s'agira d'une clé étrangère qui pointera vers la colonne `NAME`.

Si l'application évolue et qu'elle doive prendre en compte toutes les ligues sportives, nous risquons de nous retrouver avec des doublons de `name`. Nous serions alors obligés de redéfinir cette clé primaire comme étant la combinaison de la colonne `NAME` et de la

colonne `ID_LIGUE`, qui est bien entendu elle aussi une clé étrangère vers la toute nouvelle table `LIGUE`. Malheureusement, nous avons déjà plusieurs clés étrangères qui pointent vers l'ancienne clé primaire, notamment dans `COACH` et `PLAYER`, alors que notre application ne compte pas plus d'une dizaine de tables.

Nous voyons bien que ce type de maintenance devient vite coûteux. De plus, à l'issue de cette modification, nous aurions à travailler avec une clé composée, qui est plus délicate à gérer à tous les niveaux.

Pour éviter de telles complications, la notion de *clé artificielle* est adoptée par beaucoup de concepteurs d'applications. Contrairement à la clé primaire précédente, la clé artificielle, ou *surrogate key,* n'a aucun sens métier. Elle présente en outre l'avantage de pouvoir être générée automatiquement, par exemple, *via* une séquence si vous travaillez avec une base de données Oracle.

Pour en savoir plus

Voici deux liens intéressants sur les *surrogate keys* : *http://en.wikipedia.org/wiki/Surrogate_key.*

La best practice en la matière consiste à définir la clé primaire de vos tables par une clé artificielle générée et d'assurer la cohérence des données à l'aide d'une contrainte d'unicité sur une colonne métier ou une combinaison de colonnes métier. Toutes vos clés étrangères sont ainsi définies autour des clés artificielles, engendrant une maintenance facile et rapide en cas d'évolution du modèle physique de données.

Pour vos nouvelles applications

Lors de l'analyse fonctionnelle, il est indispensable de recenser les données ou combinaisons de données candidates à l'unicité métier.

Couche de persistance et objet Java

En Java, l'identité peut devenir ambiguë lorsque vous travaillez avec deux objets, et il n'est pas évident de savoir si ces deux objets sont identiques techniquement ou au sens métier.

Dans l'exemple suivant :

```
Integer a = new Integer(1) ;
Integer b = new Integer(1) ;
```

l'expression `a = = b`, qui teste l'identité, n'est pas vérifiée et renvoie `false`. Pour que `a = = b`, il faut que ces deux pointeurs pointent vers le même objet en mémoire. Dans le même temps, nous souhaiterions considérer ces deux instances comme égales sémantiquement.

La méthode non finale `equals()` de la classe Object permet de redéfinir la notion d'égalité de la façon suivante (il s'agit bien d'une redéfinition, car, par défaut, une instance n'est égale qu'à elle-même) :

```
public boolean equals(Object o){
return (this = = o);
}
```

Dans cet exemple, si nous voulons que les expressions `a` et `b` soient égales, nous surchargeons `equals()` en :

```
Public boolean equals(Object o){
if((o!=null) && (obj instanceof Integer)){
    return ((Integer)this).intValue() = = ((Integer)obj).intValue();
  }
  return false;
}
```

La notion d'identité (de référence en mémoire) est délaissée au profit de celle d'égalité, qui est indispensable lorsque vous travaillez avec des objets dont les valeurs forment une partie de votre problématique métier.

Le comportement désiré des clés d'une map ou des éléments d'un set dépend essentiellement de la bonne implémentation d'`equals()`.

Pour en savoir plus

Les deux références suivantes vous permettront d'appréhender entièrement cette problématique :

http://developer.java.sun.com/developer/Books/effectivejava/Chapter3.pdf.

http://www-106.ibm.com/developerworks/java/library/j-jtp05273.html.

Pour redéfinir `x.equals(y)`, procédez de la façon suivante :

1. Commencez par tester `x = = y`. Il s'agit d'optimiser et de court-circuiter les traitements suivants en cas de résultat positif.
2. Utilisez l'opérateur `instanceof`. Si le test est négatif, retournez false.
3. Castez l'objet y en instance de la classe de x. L'opération ne peut être qu'une réussite étant donné le test précédent.
4. Pour chaque propriété métier candidate, c'est-à-dire celles qui garantissent l'unicité, testez l'égalité des valeurs.

Si vous surchargez `equals()`, il est indispensable de surcharger aussi `hashcode()` afin de respecter le contrat d'Object. Si vous ne le faites pas, vos objets ne pourront être stables en cas d'utilisation de collections de type HashSet, HashTable ou HashMap.

Importance de l'identité

Il existe deux cas où ne pas redéfinir `equals()` risque d'engendrer des problèmes : lorsque vous travaillez avec des clés composées et lorsque vous testez l'égalité de deux entités provenant de deux gestionnaires d'entités différents.

Votre réflexe serait dans ces deux cas d'utiliser la propriété mappée `id`. Les références mentionnées à la section « Pour en savoir plus » précédente décrivent aussi le contrat de `hashcode()`. Vous y apprendrez notamment que le moment auquel le `hashcode()` est appelé importe peu, la valeur retournée devant toujours être la même. C'est la raison pour laquelle, vous ne pouvez vous fonder sur la propriété mappée `id`. En effet, cette propriété n'est renseignée qu'à l'appel de `entityManager.persist(obj)`, que vous découvrirez au chapitre 3. Si, avant cet appel, vous avez stocké votre objet dans un HashSet, le contrat est rompu puisque le `hashcode()` a changé.

> ***equals()* et *hashcode()* sont-ils obligatoires ?**
>
> Si vous travaillez avec des composite-id, vous êtes obligé de surcharger les deux méthodes au moins pour ces classes. Si vous faites appel deux fois à la même entité mais ayant été obtenue par des gestionnaires d'entités différents, vous êtes obligé de redéfinir les deux méthodes. L'utilisation de l'id dans ces méthodes est source de bogue, tandis que celle de l'unicité métier couvre tous les cas d'utilisation.
>
> Si vous n'utilisez pas les composite-id et que vous soyez certain de ne pas mettre en concurrence une même entité provenant de deux gestionnaires d'entités différents, il est inutile de vous embêter avec ces méthodes.
>
> Notez que, depuis Hibernate 3, le moteur de persistance Hibernate est beaucoup moins sensible à l'absence de redéfinition de ces méthodes. Les écrire est cependant toujours recommandé.

Les méthodes métier

Les méthodes métier de l'exemple suivant peuvent paraître évidentes, mais ce sont bien elles qui permettent de dire que votre modèle est réellement orienté objet car tirant profit de l'isolation et de la réutilisabilité :

```
/**
* @return retourne le nombre de game à score null.
*/
public int getNbNull() {
  // un simple calcul pour avoir le nombre de match nul
  return nbPlayed - nbLost - nbWon;
}

/**
    * @return la liste des games gagnés
    */
public Set getWonGames(){
  games = getGames();
  wonGames.clear();
  for (Iterator it=games.values().iterator(); it.hasNext(); ) {
    Game game = (Game)it.next();
```

```
            // si l'équipe ayant gagné le match est l'entité elle-même
        // alors le match peut aller dans la collection des matchs
        // gagnés
        if (game.getVictoriousTeam().equals(this))
          wonGames.add(game);
      }
      return wonGames;
    }
```

Sans cet effort d'enrichissement fonctionnel dans les classes composant votre modèle de classes métier, lors des phases de conception, ces logiques purement métier sont déportées dans la couche contrôle, voire la couche service. Vous vous retrouvez dès lors avec un modèle métier anémique, du code dupliqué et une maintenance et une évolution délicates.

Pour en savoir plus

L'article suivant décrit parfaitement le symptôme du modèle anémique : *http://www.martinfowler.com/bliki/AnemicDomainModel.html*

Cycle de vie d'un objet manipulé avec le gestionnaire d'entités

Pour être persistant, un objet doit pouvoir être stocké sur un support lui garantissant une durée de vie potentiellement infinie. Le plus simple des supports de stockage est un fichier qui se loge sur un support physique. La *sérialisation* permet, entre autres, de transformer un objet en fichier.

Java Persistence ne peut se brancher sur une base de données objet mais peut travailler avec n'importe quelle base de données qui dispose d'un pilote JDBC de qualité, tout du moins pour ce qui est de l'implémentation Hibernate.

Les concepts que nous allons aborder ici sont communs à toutes les solutions de mapping objet-relationnel, ou ORM (Object Relational Mapping), fondées sur la notion d'état.

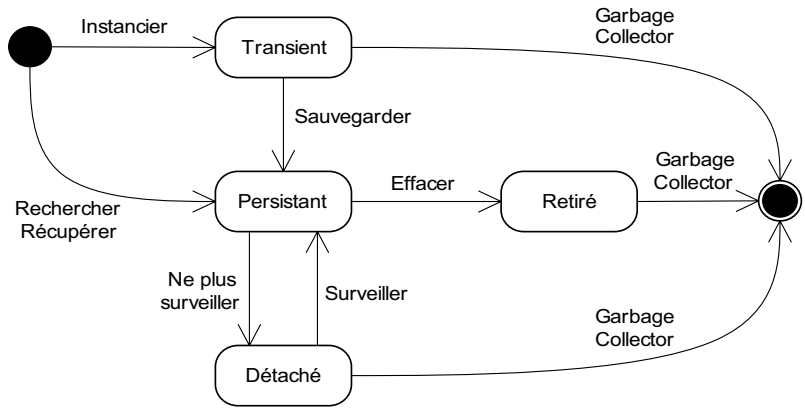
La figure 2.5 illustre les différents états d'un objet. Les états définissent le cycle de vie d'un objet.

Un objet *persistant* est un objet qui possède son image dans le datastore et dont la durée de vie est potentiellement infinie. Pour garantir que les modifications apportées à un objet sont rendues persistantes, c'est-à-dire sauvegardées, l'objet est surveillé par un « traqueur » d'instances persistantes. Ce rôle est joué par le gestionnaire d'entités.

Un objet *transient* est un objet qui n'a pas son image stockée dans le datastore. Il s'agit d'un objet « temporaire », qui meurt lorsqu'il n'est plus utilisé par personne. En Java, le garbage collector le ramasse lorsque aucun autre objet ne le référence.

Un objet *détaché* est un objet qui possède son image dans le datastore mais qui échappe temporairement à la surveillance opérée par le gestionnaire d'entités. Pour que les modifications potentiellement apportées pendant cette phase de détachement soient enregistrées, il faut effectuer une opération manuelle pour *merger* cette instance au gestionnaire d'entités.

Figure 2-5
États d'un objet



Un objet *retiré* est un objet actuellement géré par le gestionnaire d'entités mais programmé pour ne plus être persistant. À la validation de l'unité de travail, un ordre SQL delete sera exécuté pour retirer son image du datastore.

Entités et valeurs

Pour comprendre le comportement, au sens Java, des différents objets dans le contexte d'un service de persistance, nous devons les séparer en deux groupes, les entités et les objets inclus (*embedded objects*, historiquement appelés composants par Hibernate).

Une *entité* existe indépendamment de n'importe quel objet contenant une référence à cette entité. C'est là une différence notable avec le modèle Java habituel, dans lequel un objet non référencé est un candidat pour le garbage collector. Les entités doivent être explicitement rendues persistantes et supprimées, excepté dans le cas où ces actions sont définies en cascade depuis un objet *parent* vers ses enfants (la notion de cascade est liée à la persistance transitive, que nous détaillons au chapitre 6). Les entités supportent les références partagées et circulaires. Elles peuvent aussi être versionnées.

Un état persistant d'une entité est constitué de références vers d'autres entités et d'instances de type valeur. Les valeurs sont des types primitifs, des collections, des objets inclus et certains objets immuables. Contrairement aux entités, les valeurs sont rendues persistantes et supprimées par référence *(reachability)*.

Puisque les objets de types valeurs et primitifs sont rendus persistants et supprimés en relation avec les entités qui les contiennent, ils ne peuvent être versionnés indépendamment de ces dernières. Les valeurs n'ont pas d'identifiant indépendant et ne peuvent donc être partagées entre deux entités ou collections.

Tous les types pris en charge par Java Persistence, à l'exception des collections, supportent la sémantique null.

Jusqu'à présent, nous avons utilisé les termes *classes persistantes* pour faire référence aux entités. Nous allons continuer de le faire. Cependant, dans l'absolu, toutes les classes

persistantes définies par un utilisateur et ayant un état persistant ne sont pas nécessairement des entités. Un composant, par exemple, est une classe définie par l'utilisateur ayant la sémantique d'une valeur.

En résumé

Nous venons de voir les éléments structurant une classe persistante et avons décrit leur importance en relation avec Java Persistence. Nous avons par ailleurs évoqué les différences entre les notions d'entité et de valeur.

La définition du cycle de vie d'un objet persistant manipulé avec Java Persistence a été abordée, et vous savez déjà que le gestionnaire d'entités vous permettra de faire vivre vos objets persistants selon ce cycle de vie.

Les bases sont posées pour appréhender concrètement l'utilisation du gestionnaire d'entités. À chaque transition du cycle de vie correspond au moins une méthode à invoquer sur le gestionnaire. C'est ce que nous nous proposons de décrire à la section suivante.

Le gestionnaire d'entités

L'utilisation du gestionnaire d'entité ne peut se faire sans les métadonnées. À l'inverse, les métadonnées ne sont testables et donc compréhensibles sans la connaissance de l'API EntityManager.

L'utilisation atomique du gestionnaire d'entités dans nos exemples suit le modèle suivant :

```
EntityManager em =
    (EntityManager) new InitialContext()
       .lookup("java:/EntityManagers/eyrollesEntityManager"); ←❶

TransactionManager tm =
    (TransactionManager) new InitialContext()
       .lookup("java:/TransactionManager");

tm.begin(); ←❷
//faire votre travail←❸
    ...
tm.commit();
```

Comme expliqué précédemment, le gestionnaire d'entités, s'obtient pour le moment *via* le registre JNDI (repère ❶). La gestion de transaction (repère ❷) est indispensable pour vous permettre d'utiliser le gestionnaire d'entités, même s'il est possible de la rendre partiellement automatique et transparente, comme vous le verrez ultérieurement avec l'utilisation des beans session et de la gestion déclarative des transactions.

Souvenez-vous de vous référer au chapitre 7 pour appréhender la problématique de gestion des exceptions.

Les actions du gestionnaire d'entités

Vous allez maintenant vous intéresser de manière très générale aux actions que vous pouvez effectuer à partir d'un gestionnaire d'entités. À partir du même exemple de code, vous insérerez (repère ❸) les exemples de code fournis dans les sections qui suivent.

Les actions abordées ci-après ne couvrent pas l'intégralité des possibilités qui vous sont offertes. Pour une même action, il peut en effet exister deux ou trois variantes. Une description exhaustive des actions entraînant une écriture en base de données est fournie au chapitre 6.

Récupération d'une entité

Pour récupérer une entité, il existe plusieurs façons de procéder.

Si vous connaissez son id, invoquez `em.find` (Class clazz, Object id). En cas de non-existence de l'objet, null sera retourné :

```
Player player = (Player) em.find (Player.class, new Long(1));
select player0_.PLAYER_ID as PLAYER_ID0_, player0_.PLAYER_NAME as
PLAYER_N2_0_0_, player0_.PLAYER_NUMBER as PLAYER_N3_0_0_,
player0_.BIRTHDAY as BIRTHDAY0_0_, player0_.HEIGHT as HEIGHT0_0_,
player0_.WEIGHT as WEIGHT0_0_, player0_.TEAM_ID as TEAM_ID0_0_
from PLAYER player0_
where player0_.PLAYER_ID=?
```

Cette méthode permet la récupération unitaire d'une entité. Vous pouvez bien sûr former des requêtes orientées objet, *via* le langage EJB-QL, par exemple. Une dernière possibilité consiste à récupérer virtuellement l'instance *via* la méthode `em.getReference` (Class clazz, Object id). Dans ce cas, le gestionnaire d'entités ne consulte pas la base de données mais vous renvoie un proxy. Dès que vous tenterez d'accéder à l'état de cet objet, en invoquant un accesseur autre que l'id, la base de données sera interrogée pour « remplir » l'instance.

Rendre une nouvelle instance persistante

La méthode `em.persist()` permet de rendre persistante une instance transiente, par exemple une nouvelle instance, l'instanciation pouvant se faire à n'importe quel endroit de l'application :

```
Player player = new Player("Zidane") ;
em.persist(player);
//commit
insert into PLAYER (PLAYER_NAME, PLAYER_NUMBER, BIRTHDAY, HEIGHT, WEIGHT,
TEAM_ID) values (?, ?, ?, ?, ?, ?)
```

Vous pouvez aussi utiliser `em.merge()`.

Rendre persistantes les modifications d'une instance

Si l'instance persistante est présente dans le gestionnaire d'entités, il n'y a rien à faire. Le simple fait de la modifier engendre une mise à jour lors du commit ; c'est ce qu'on appelle le *dirty checking* automatique :

```
Player player = (Player) em.find(Player.class, new Long(1));
player.setName("zidane");
Hibernate: select player0_.PLAYER_ID as PLAYER_ID0_,
player0_.PLAYER_NAME as PLAYER_N2_0_0_,
player0_.PLAYER_NUMBER as PLAYER_N3_0_0_,
player0_.BIRTHDAY as BIRTHDAY0_0_,
player0_.HEIGHT as HEIGHT0_0_, player0_.WEIGHT as WEIGHT0_0_,
player0_.TEAM_ID as TEAM_ID0_0_
from PLAYER player0_
where player0_.PLAYER_ID=?
Hibernate: update PLAYER set PLAYER_NAME=?, PLAYER_NUMBER=?, BIRTHDAY=?,
HEIGHT=?, WEIGHT=?, TEAM_ID=? where PLAYER_ID=?
```

Ici, le select est le résultat de la première ligne de code, et l'update se déclenche au commit de la transaction, ou plutôt à l'appel de `flush()`. Le flush est une notion importante, sur laquelle nous reviendrons plus tard. Sachez simplement pour l'instant que, par défaut, Java Persistence, exécute automatiquement un flush au bon moment afin de garantir la consistance des données.

Rendre persistantes les modifications d'une instance détachée

Si l'instance persistante n'est pas liée à un gestionnaire d'entités, elle est dite *détachée*. Il est impossible de traquer les modifications des instances persistantes qui ne sont pas attachées à un gestionnaire d'entités. C'est la raison pour laquelle le détachement et le réattachement font partie du cycle de vie de l'objet du point de vue de la persistance.

Il est impossible de réattacher à proprement parlé une instance détachée. La seule action définie par Java Persistence est appelée *merge*. En invoquant merge sur un gestionnaire d'entités, celui-ci charge l'état persistant de l'entité et fusionne l'état persistant avec l'état détaché.

Pour merger et rendre persistantes les modifications qui auraient pu survenir en dehors du scope du gestionnaire d'entités, il suffit d'invoquer la méthode `em.merge (entitéDétachée)` :

Le premier select est déclenché à l'invocation de `em.merge()`. Java Persistence récupère les données et les fusionne avec les modifications apportées à l'instance détachée. `em.merge()`

```
Player attachedPlayer = (Player)em.merge(detachedPlayer);
Hibernate: select player0_.PLAYER_ID as PLAYER_ID0_,
player0_.PLAYER_NAME as PLAYER_N2_0_0_,
player0_.PLAYER_NUMBER as PLAYER_N3_0_0_,
player0_.BIRTHDAY as BIRTHDAY0_0_, player0_.HEIGHT as HEIGHT0_0_,
player0_.WEIGHT as WEIGHT0_0_, player0_.TEAM_ID as TEAM_ID0_0_
from PLAYER player0_
where player0_.PLAYER_ID=?
Hibernate: update PLAYER set PLAYER_NAME=?, PLAYER_NUMBER=?, BIRTHDAY=?,
  HEIGHT=?, WEIGHT=?, TEAM_ID=? where PLAYER_ID=?
```

renvoie alors l'instance persistante, et celle-ci est attachée au gestionnaire d'entités. À l'issue de l'exécution de ce code, vous avez bien `detachedPlayer` qui est toujours détaché, mais `attachedPlayer` comprend quant à lui les modifications et est attaché.

Détacher une instance persistante

Détacher une instance signifie ne plus surveiller cette instance, avec les deux conséquences majeures suivantes :

- Plus aucune modification ne sera rendue persistante de manière transparente.
- Tout contact avec un proxy engendrera une erreur.

Nous reviendrons dans le cours de l'ouvrage sur la notion de proxy. Sachez simplement qu'un proxy est nécessaire pour l'accès à la demande, ou *lazy loading (voir plus loin),* des objets associés.

Il existe trois moyens de détacher une instance :

- en fermant le gestionnaire d'entités : `em.close()` ;
- en le vidant : `em.clear()` ;
- (spécifique d'Hibernate), en détachant une instance particulière : `hibernateSession.evict(obj)`.

Enlever un objet

Enlever un objet signifie l'extraire définitivement de la base de données. La méthode `em.remove()` permet d'effectuer cette opération. Prenez garde cependant que l'enregistrement n'est dès lors plus présent en base de données et que l'instance reste dans la JVM tant que l'objet est référencé. S'il ne l'est plus, il est ramassé par le garbage collector :

```
em.remove(player);
delete from PLAYER where PLAYER_ID=?
```

Rafraîchir une instance

Dans le cas où un trigger serait déclenché suite à une opération (`ON INSERT`, `ON UPDATE`, etc.), vous pouvez forcer le rafraîchissement de l'instance *via* `em.refresh()`.

Cette méthode déclenche un select et met à jour les valeurs des propriétés de l'instance.

Exercices

Pour chacun des exemples de code ci-dessous, définissez l'état de l'instance de `Player`, en supposant que le gestionnaire d'entités est vide.

Énoncé 1

```
public void test1(Player p){
←❶
  …
  tm.begin();
  EntityManager em = (EntityManager) new InitialContext()
     .lookup("java:/EntityManagers/eyrollesEntityManager");
  tm.commit();
  ←❷
}
```

Solution

En ❶, l'instance provient d'une couche supérieure. Émettons l'hypothèse qu'elle est détachée. Un gestionnaire d'entités est ensuite récupéré, mais cela ne suffit pas. En ❷, l'instance est toujours détachée.

Énoncé 2

```
public Player test2(Long id){
// nous supposons que l'id existe dans le datastore
  …
  tm.begin();
  EntityManager em = (EntityManager) new InitialContext()
     .lookup("java:/EntityManagers/eyrollesEntityManager");
  Player p = em.find (Player.class,id);
  ←❶
  tm.commit();
  return p ; ←❷
}
```

Solution

L'instance est récupérée *via* le gestionnaire d'entités. Elle est donc attachée jusqu'à fermeture ou détachement explicite. Dans ce test, l'instance est persistante (et attachée) en ❶ et ❷.

Énoncé 3

```
public Team test3(Long id){
  …
  tm.begin();
  EntityManager em = (EntityManager) new InitialContext()
```

```
      .lookup("java:/EntityManagers/eyrollesEntityManager");
  Player p = em.find (Player.class,id);
  ←❶
  tm.commit();
  em.close();
  return p.getTeam(); ←❷
}
```

Solution

En ❶, l'instance est persistante, mais elle est détachée en ❷ car le gestionnaire d'entités est fermé. La ligne pourra soulever une exception de chargement, mais, pour l'instant, vous n'êtes pas en mesure de savoir pourquoi. Vous le verrez au chapitre 5.

Énoncé 4

```
public void test4(Player p){
// nous supposons que p.getId() existe dans le datastore
  …
  tm.begin();
  EntityManager em = (EntityManager) new InitialContext()
     .lookup("java:/EntityManagers/eyrollesEntityManager");

  ←❶
  em.remove (p);
  ←❷
  tm.commit();
  em.close();
}
```

Solution

L'instance est détachée puis transiente.

Énoncé 5

```
public void test5(Player p){
// nous supposons que p.getId() existe dans le datastore
  …
  tm.begin();
  EntityManager em = (EntityManager) new InitialContext()
     .lookup("java:/EntityManagers/eyrollesEntityManager");
  ←❶
  p = em.merge(p);
  ←❷
  tm.commit();
  em.close();
}
```

Solution

L'instance est détachée puis persistante. Pour autant, que pouvons-nous dire de `p.getTeam()` ? Vous serez en mesure de répondre à cette question après avoir lu le chapitre 6.

Énoncé 6

```
public void test7(Player p){
  …
  tm.begin();
  EntityManager em = (EntityManager) new InitialContext()
     .lookup("java:/EntityManagers/eyrollesEntityManager");
  ←❶
  Player p2 = em.merge(p);
  ←❷
  tm.commit();
  em.close();
}
```

Solution

En ❶, l'instance `p` peut être transiente ou détachée. En ❷, `p` est détachée et `p2` persistante.

En résumé

Nous sommes désormais en mesure de compléter le diagramme d'états *(voir figure 2.6)* que nous avons esquissés à la figure 2.5 avec les méthodes du gestionnaire d'entités permettant la transition d'un état à un autre ainsi que les méthodes, si vous utilisez Hibernate nativement *(voir figure 2.7)*.

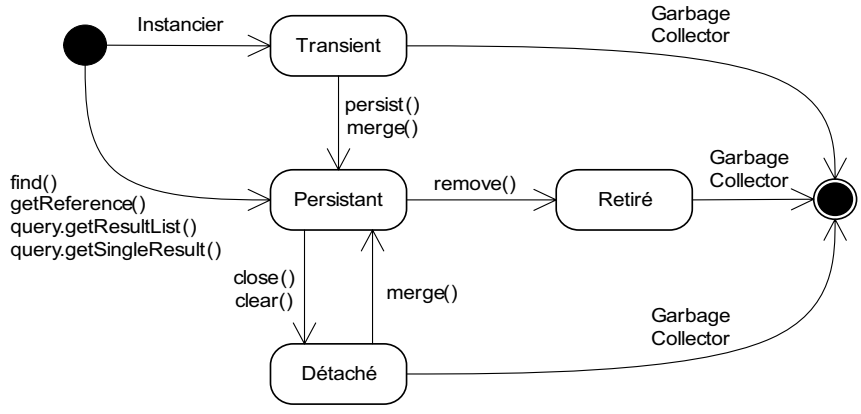


Figure 2-6
Cycle de vie des instances persistantes avec Java Persistence

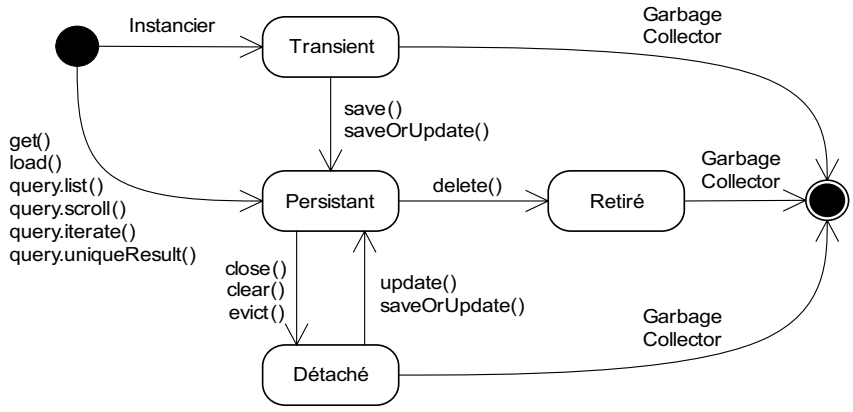


Figure 2-7
Cycle de vie des instances persistantes avec Hibernate natif

Conclusion

Vous disposez maintenant des informations nécessaires à la mise en place de l'environnement de prototypage utilisé tout au long de ce livre. Vous connaissez aussi les subtilités du cycle de vie des objets dans le cadre de l'utilisation d'un outil de persistance fondé sur la notion d'état et êtes capable de mettre un nom de méthode sur chaque transition de ce cycle de vie.

Sur le plan théorique, il ne vous reste plus qu'à connaître les métadonnées, qui vous permettront de mapper vos entités à votre schéma relationnel. C'est ce que nous vous proposons d'aborder au chapitre 3.