2

Classes persistantes et session Hibernate

Le rôle d'un outil de mapping objet-relationnel tel qu'Hibernate est de faire abstraction du schéma relationnel qui sert à stocker les objets métier. Les fichiers de mapping permettent de faire le lien entre votre modèle de classes métier et le schéma relationnel. Avec Hibernate, le développeur d'application n'a plus à se soucier *a priori* de la base de données. Il lui suffit d'utiliser les API d'Hibernate, notamment l'API `Session`, 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'objet. 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 de l'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 instance persistante passe par la session Hibernate. Ce chapitre introduit les éléments constitutifs d'une session Hibernate et détaille ce qui définit une classe métier persistante.

Avant de découvrir les actions que vous pouvez mener sur des objets persistants depuis la session et leur impact sur la vie des objets, vous commencerez par installer et configurer Hibernate afin de pouvoir tester les exemples de code fournis.

Installation d'Hibernate

Cette section décrit les étapes minimales permettant de mettre en place un environnement de développement Hibernate susceptible de gérer une application.

Dans une telle configuration de base, Hibernate est simple à installer et ne nécessite d'autre paramétrage que celui du fichier **hibernate.cfg.xml.**

Les bibliothèques Hibernate

Lorsque vous téléchargez Hibernate, vous 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.

Tableau 2.1. Bibliothèques d'Hibernate

Catégorie	Bibliothèque	Fonction	Particularité
Indispensables à l'exécution	`dom4j-1.5.2.jar (1.5.2)`	Parseur de configuration XML et de mapping	Exécution. Requis
	`xml-apis.jar (unknown)`	API standard JAXP	Exécution. Un parser SAX est requis.
	`commons-logging-1.0.4.jar (1.0.4)`	Commons Logging	Exécution. Requis
	`jta.jar (unknown)`	API Standard JTA	Exécution. Requis pour les applications autonomes s'exécutant en dehors d'un serveur d'applications
	`jdbc2_0-stdext.jar (2.0)`	API JDBC des extensions standards	Exécution. Requis pour les applications autonomes s'exécutant en dehors d'un serveur d'applications
	`antlr-2.7.4.jar (2.7.4)`	ANTLR (ANother Tool for Language Recognition)	Exécution
	`cglib-full-2.0.2.jar (2.0.2)`	Générateur de bytecode CGLIB	Exécution. Requis
	`xerces-2.6.2.jar (2.6.2)`	Parser SAX	Exécution. Requis
	`commons-collections-2.1.1.jar (2.1.1)`	Collections Commons	Exécution. Requis
En relation avec le pool de connexions	`c3p0-0.8.5.jar (0.8.5)`	Pool de connexions JDBC C3P0	Exécution. Optionnel
	`proxool-0.8.3.jar (0.8.3)`	Pool de connexions JDBC Proxool	Exécution. Optionnel
En relation avec le cache	`ehcache-1.1.jar (1.1)`	Cache EHCache	Exécution. Optionnel. Requis si aucun autre fournisseur de cache n'est paramétré.
Autres bibliothèques optionnelles	`jboss-cache.jar (1.2)`	Cache en clusters TreeCache	Exécution. Optionnel
	`jboss-system.jar (unknown)`		Exécution. Optionnel. Requis par TreeCache
	`jboss-common.jar (unknown)`		Exécution. Optionnel. Requis par TreeCache
	`jboss-jmx.jar (unknown)`		Exécution. Optionnel. Requis par TreeCache

Tableau 2.1. Bibliothèques d'Hibernate

Catégorie	Bibliothèque	Fonction	Particularité
Autres bibliothèques optionnelles	`concurrent-1.3.2.jar (1.3.2)`		Exécution. Optionnel. Requis par TreeCache
	`jboss-remoting.jar (unknown)`		Exécution. Optionnel. Requis par TreeCache
	`swarmcache-1.0rc2.jar (1.0rc2)`		Exécution. Optionnel
	`jgroups-2.2.7.jar (2.2.7)`	Bibliothèque multicast JGroups	Exécution. Optionnel. Requis par les caches supportant la réplication
	`oscache-2.1.jar (2.1)`	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-1_0-fr.jar (1.0-fr)`	Bibliothèque JACC	Exécution. Optionnel
	`log4j-1.2.9.jar (1.2.9)`	Bibliothèque Log4j	Exécution. Optionnel
	`jaxen-1.1-beta-4.jar (1.1-beta-4)`	Jaxen, moteur XPath Java universel	Exécution. Requis si vous souhaitez désérialiser. Configuration pour améliorer les performances au démarrage.
Indispensables pour recompiler Hibernate	`versioncheck.jar (1.0)`	Vérificateur de version	Compilation
	`checkstyle-all.jar`	Checkstyle	Compilation
	`junit-3.8.1.jar (3.8.1)`	Framework de test JUnit	Compilation
	`ant-launcher-1.6.2.jar (1.6.2)`	Launcher Ant	Compilation
	`ant-antlr-1.6.2.jar (1.6.2)`	Support ANTLR Ant	Compilation
	`cleanimports.jar (unknown)`	Cleanimports	Compilation
	`ant-junit-1.6.2.jar (1.6.2)`	Support JUnit-Ant	Compilation
	`ant-swing-1.6.2.jar (1.6.2)`	Support Swing-Ant	Compilation
	`ant-1.6.2.jar (1.6.2)`	Core Ant	ant-1.6.2.jar (1.6.2)

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.

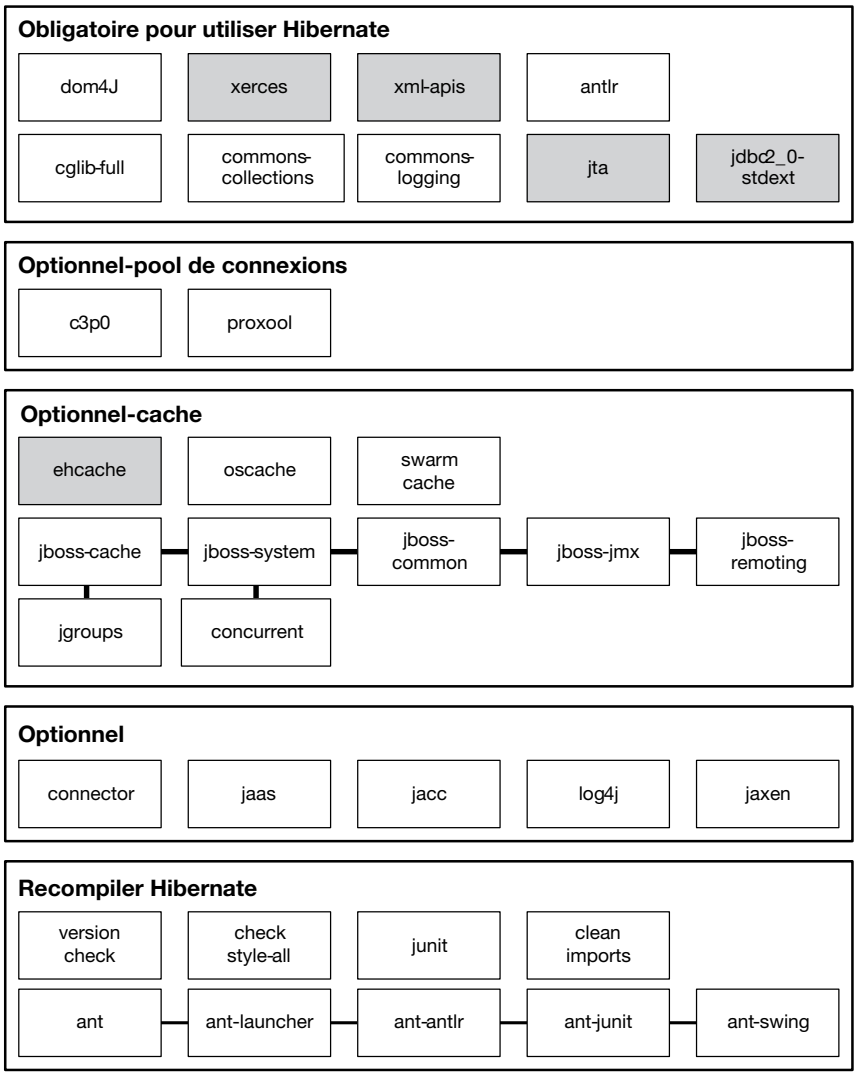


Figure 2.1
Les bibliothèques Hibernate

Le fichier de configuration globale hibernate.cfg.xml

Une fois les bibliothèques, dont **hibernate3.jar** et le pilote JDBC, en place, vient l'étape de configuration des informations globales nécessaires au lancement d'Hibernate.

Le fichier **hibernate.cfg.xml** regroupe toutes les informations concernant les classes persistantes et l'intégration d'Hibernate avec les autres composants techniques de l'application, notamment les suivants :

- source de donnée (`datasource` ou `jdbc`) ;
- pool de connexions (par exemple, `c3p0` si la source de données est `jdbc`) ;

- cache de second niveau (par exemple, `EHCache`) ;
- affichage des traces (avec log4j) ;
- configuration de la notion de batch.

Vous allez effectuer une première configuration d'Hibernate permettant de le manipuler rapidement. Il vous faut télécharger pour cela le pilote JDBC (`connection.driver_class`) et le placer dans le classpath de l'application.

Pour vos premiers pas dans Hibernate, il est préférable d'utiliser une base de données gratuite et facile à mettre en place, telle que HSQLDB *(http://hsqldb.sourceforge.net/) ou* MySQL avec innoDB *(http://www.mysql.com/).* Dans ce dernier cas, le pilote à télécharger est `mysql- connector-java-3.0.15-ga-bin.jar`.

Gestion des transactions avec MySQL

Les anciennes versions de MySQL ne supportent pas les transactions. Or sans transaction, il est impossible d'utiliser Hibernate convenablement. Il faut activer innoDB pour obtenir le support des transactions *(http://www.sourcekeg.co.uk/www.mysql.com/doc/mysql/fr/InnoDB_news-4.0.20.html).*

Comme expliqué précédemment, la performance de la couche de persistance est directement liée à la qualité des pilotes JDBC employés. Pensez à exiger de la part de votre fournisseur une liste des bogues recensés, afin d'éviter de perdre de longues heures en cas de bogue dans le code produit. Par exemple, sous Oracle, vous obtenez un bogue si vous utilisez `Lob` avec les pilotes officiels (il existe cependant des moyens de contourner ce bogue).

Voici le fichier de votre configuration simple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
  PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="dialect">
      net.sf.hibernate.dialect.MySQLDialect</property>
    <property name="connection.driver_class">
      org.gjt.mm.mysql.Driver</property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost/SportTracker</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
    <property name="show_sql">true</property>
    <!-- Mapping files -->
    <mapping resource="com/eyrolles/sportTracker/model/Player.hbm.xml"/>
    <mapping resource="com/eyrolles/sportTracker/model/Game.hbm.xml"/>
    <mapping resource="com/eyrolles/sportTracker/model/Team.hbm.xml"/>
    <mapping resource="com/eyrolles/sportTracker/model/Coach.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Dans ce fichier, aucune notion de pool de connexions n'apparaît. Les valeurs par défaut sont une gestion de pool *via* `DriverManagerConnectionProvider` pour un pool de vingt connexions au maximum.

Ce type de configuration convient à un environnement de développement mais ne peut être utilisé en production, comme indiqué dans les logs générés ci-dessous :

```
INFO DriverManagerConnectionProvider:41 - Using Hibernate built-in connection pool (not
for production use!)
INFO DriverManagerConnectionProvider:42 - Hibernate connection pool size: 20
INFO DriverManagerConnectionProvider:45 - autocommit mode: false
```

Vous pouvez utiliser à la place un fichier de propriétés **hibernate.properties,** mais il est recommandé de travailler avec la version XML, qui permet un paramétrage plus fin. En effet, toutes les possibilités de paramétrage ne sont pas disponibles *via* le fichier **hibernate.properties,** comme la définition du cache pour les entités.

Les tableaux 2.2 à 2.6 donnent la liste des paramètres du fichier **hibernate.cfg.xml.**

Le tableau 2.2 récapitule les paramètres JDBC à mettre en place. Utilisez-les si vous ne disposez pas d'une datasource. Ce paramétrage nécessite de choisir un pool de connexions. Hibernate est livré avec C3P0 et Proxool. Référez-vous au chapitre 8 pour les détails de configuration des pools de connexions.

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

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.

Tableau 2.3. Paramétrage de la datasource

Paramètre	Rôle
`hibernate.connection.datasource`	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.

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`
Oracle 9/10g	`org.hibernate.dialect.Oracle9Dialect`
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 transaction disponibles en fonction du serveur d'applications utilisé.

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. Pour en savoir plus sur ces paramètres, référez-vous au guide de référence.

Tableau 2.6. 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.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).
`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.

Tableau 2.6. Paramètres optionnels

Paramètre	Rôle
`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.

Votre configuration d'Hibernate est maintenant opérationnelle. Vous ferez momentanément abstraction des fichiers de mapping, qui permettent de mettre en correspondance vos classes Java et votre modèle relationnel. Ces fichiers sont abordés à la section suivante. Considérez pour le moment qu'ils se trouvent dans le classpath.

Les composants de l'architecture d'Hibernate (hors session)

Les principaux composants indispensables à l'obtention d'une session Hibernate sont illustrés à la figure 2.2.

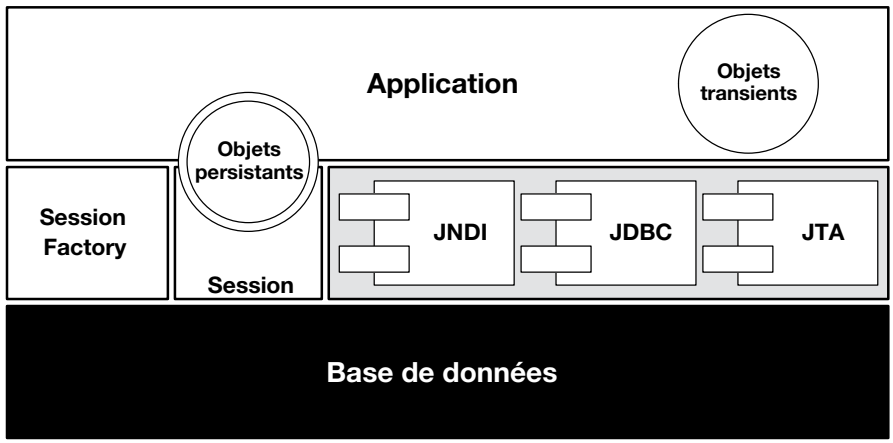


Figure 2.2
Composants de l'architecture d'Hibernate

Cette architecture peut être résumée de la façon suivante : votre application dispose d'objets, dont la persistance est gérée par une session Hibernate. Rappelons qu'un objet est dit persistant lorsque sa durée de vie est longue, à l'inverse d'un objet transient, qui est temporaire.

Une session s'obtient *via* une `SessionFactory` ; la `SessionFactory` est construite à partir d'un objet `Configuration` et contient les informations de configuration globale ainsi que les informations contenues dans les fichiers de mapping, appelées *métadonnées.*

Une session effectue des opérations, dont la plupart ont un lien direct avec la base de données. Ces opérations se déroulent au sein d'une *transaction* et reposent sur une connexion JBDC. Cette connexion est fournie par le `ConnectionProvider`.

Hibernate repose essentiellement sur JDBC mais peut aussi utiliser JNDI et JTA pour certains environnements.

Les trois étapes suivantes permettent d'obtenir une nouvelle session :

1. Instanciation de l'objet `Configuration` en appelant le constructeur `new Configuration()`. Cet appel consulte le fichier **hibernate.cfg.xml** présent dans le classpath de l'application.
2. Construction de la `SessionFactory` *via* la méthode `configuration.configure().buildSessionFactory()`. L'analyse des fichiers de mapping a lieu à ce moment.
3. Demande de nouvelle session à la `SessionFactory` en invoquant `sessionFactory.openSession()`.

Les objets `Configuration` et `SessionFactory` sont coûteux à instancier puisqu'ils requièrent, entre autres, l'analyse des fichiers de mapping présents dans votre application. Il est donc judicieux de les stocker dans un singleton (une seule instance pour l'application) en utilisant des variables statiques. Ces objets sont *threadsafe,* c'est-à-dire qu'ils peuvent être attaqués par plusieurs traitements en parallèle.

À l'inverse, la session n'est pas threadsafe, et il est possible de la récupérer sans impacter les performances de votre application.

L'exemple de code suivant met en œuvre les trois étapes décrites ci-dessus :

```
private static Configuration configuration;
private static SessionFactory sessionFactory;
private Session s;
try {
// étape 1
configuration = new Configuration();
  // étape 2
sessionFactory = configuration.configure().buildSessionFactory();
  // étape 3
s = sessionFactory.openSession();
} catch (Throwable ex) {
log.error("Building SessionFactory failed.", ex);
  throw new ExceptionInInitializerError(ex);
}
```

Les logs suivants s'affichent lors de l'exécution des deux premières étapes :

```
INFO Environment:424 - Hibernate 3.0 beta 1
INFO Environment:437 - hibernate.properties not found
INFO Environment:470 - using CGLIB reflection optimizer
INFO Environment:500 - using JDK 1.4 java.sql.Timestamp handling
INFO Configuration:1046 - configuring from resource: /hibernate.cfg.xml
INFO Configuration:1017 - Configuration resource: /hibernate.cfg.xml
INFO Configuration:419 - Mapping resource:
 com/eyrolles/sportTracker/model/Player.hbm.xml
INFO HbmBinder:442 - Mapping class:
 com.eyrolles.sportTracker.model.Player -> PLAYER
…
INFO Configuration:1193 - Configured SessionFactory: null
INFO Configuration:760 - processing collection mappings
INFO HbmBinder:1720 - Mapping collection:
 com.eyrolles.sportTracker.model.Team.players -> PLAYER
…
INFO Dialect:86 - Using dialect: org.hibernate.dialect.MySQLDialect
INFO DriverManagerConnectionProvider:80 - using driver:
 org.gjt.mm.mysql.Driver at URL: jdbc:mysql://localhost/SportTracker
INFO DriverManagerConnectionProvider:86 - connection properties:
 {user=root, password=****}
…
INFO TransactionFactoryFactory:31 - Using default transaction strategy
(direct JDBC transactions)
15:28:06,465 INFO TransactionManagerLookupFactory:33 -
   No TransactionManagerLookup configured (in JTA environment, use of read-write or
transactional second-level cache is not recommended)
…
15:28:06,512  INFO SettingsFactory:179 - Echoing all SQL to stdout
…
15:28:06,622  INFO SessionFactoryImpl:132 - building session factory
…
15:28:09,090  INFO SessionFactoryObjectFactory:82 -
  Not binding factory to JNDI, no JNDI name configured
…
```

Ces logs vous permettent d'appréhender tous les détails de configuration pris en compte par Hibernate d'une manière relativement lisible.

Il est recommandé de vous munir d'une classe utilitaire pour faire abstraction de ces étapes et les factoriser pour l'ensemble de votre application. De même, il est préférable de créer les objets `Configuration` et `SessionFactory` dans un bloc statique afin de ne l'exécuter qu'une fois. Vous verrez au chapitre 7 ce qu'apportent les classes utilitaires à la gestion de la persistance dans une application qui repose sur Hibernate. Ces classes vous simplifieront la tâche.

Pour la suite de nos démonstrations, nous supposons que la ligne de code suivante suffit à exécuter les trois étapes de manière optimisée :

```
Session session = HibernateUtil.getSession();
```

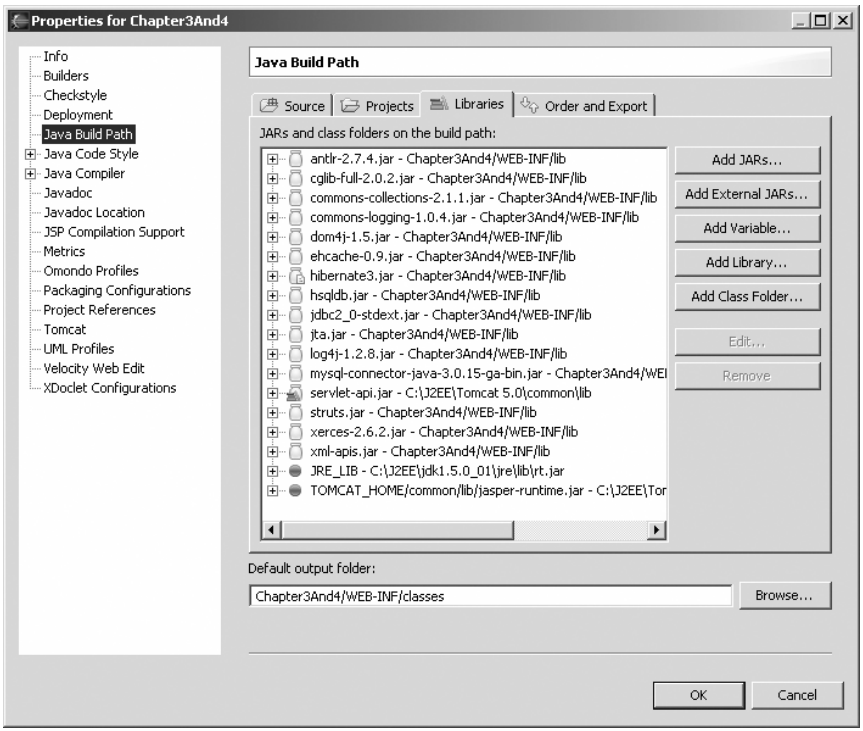
`HibernateUtil` est le nom de votre classe utilitaire et `getSession()` celui d'une méthode statique.

En résumé

Pour vos premiers pas avec Hibernate, vous disposez des prérequis pour monter un projet, par exemple, sous Eclipse, avec un chemin de compilation tel que celui illustré à la figure 2.3. Il vous suffit ensuite d'écrire le fichier **hibernate.cfg.xml** ainsi que vos fichiers de mapping.

Équipé d'une classe utilitaire qui vous fournira la session Hibernate de manière transparente, vous serez à même de manipuler le moteur de persistance Hibernate.

Figure 2.3
Exemple de projet Hibernate sous Eclipse



Les classes métier persistantes

On appelle modèle de classes métier un modèle qui définit les problématiques réelles rencontrées par une entreprise.

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.

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 une étude et une démonstration très intéressantes sur la réutilisation dans l'analyse faite par Jeffrey S. Poulin sur la page *http://home.stny.rr.com/jeffrey-poulin/Papers/Object_Mag_Metrics/oometrics.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 classes persistantes sont les classes dont les instances doivent durer dans le temps. Ce sont celles qui sont prises en compte par Hibernate. Les éléments qui composent ces classes persistantes sont décrits dans des fichiers de mapping.

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.

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 com.eyrolles.sportTracker.model;
// imports nécessaires
```

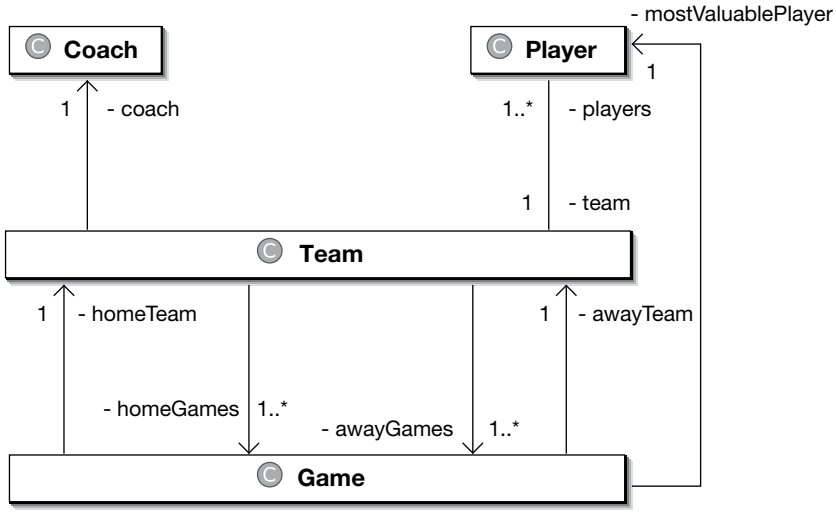
```
/**
 * Une instance de cette classe représente une Team.
* Des players et des games sont associés à cette team.
* Son cycle de vie est indépendant de celui des objets associés
* @author Anthony Patricio <anthony@hibernate.org>
 */
public class Team implements Serializable{
private Long id;←❶
  private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

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

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

Figure 2.4

Diagramme de classes exemple



Cette classe est persistante. 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 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 obligé 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 artifi-

cielle, 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://www.dbmsmag.com/9805d05.html, http://www.bcarter.com/intsurr1.htm*

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. Pour générer la clé artificielle, choisissez le générateur `native` dans vos fichiers de mapping. Hibernate fera le reste.

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 sessions différentes.

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 `session.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 sessions différentes, 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 sessions différentes, il est inutile de vous embêter avec ces méthodes.
>
> Notez qu'Hibernate 3 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*

Le cycle de vie d'un objet manipulé avec Hibernate

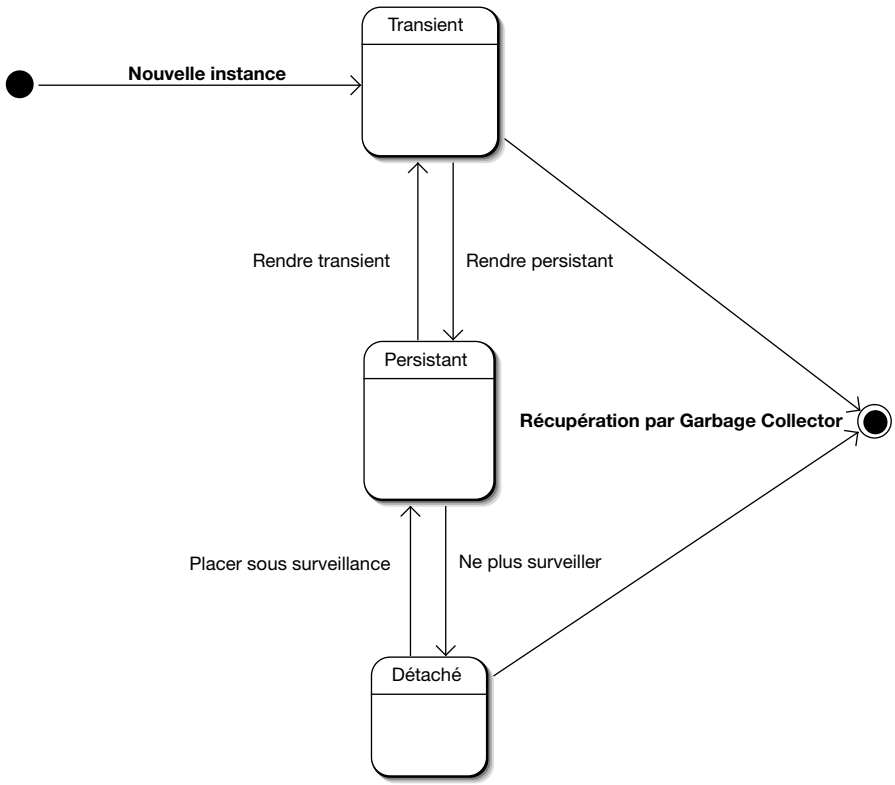
Pour être persistant, un objet doit pouvoir être stocké sur un support lui garantissant une durée de vie potentiellement infinie. Le plus simple des supports de stockage est un fichier qui se loge sur un support physique. La *sérialisation* permet, entre autres, de transformer un objet en fichier.

Hibernate 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. Une nouvelle fonctionnalité permet à Hibernate de traiter des supports XML comme source de donnée.

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.

Figure 2.5
États 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 la session dans Hibernate.

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 lorsqu'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 la session. Pour que les modifications potentiellement apportées pendant cette phase de détachement soient enregistrées, il faut réattacher manuellement cette instance à la session.

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 valeurs.

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 composants 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 Hibernate, 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 Hibernate. 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 Hibernate a été abordée, et vous savez déjà que la session Hibernate vous permettra de faire vivre vos objets persistants selon ce cycle de vie.

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

La session Hibernate

L'utilisation de la session ne peut se faire sans des fichiers de mapping. À l'inverse, les fichiers de mapping ne sont testables et donc compréhensibles sans la connaissance de l'API `Session`.

L'utilisation atomique de la session Hibernate doit suivre le schéma suivant :

```
Session session = factory.openSession();←❶
// ou Session session = HibernateUtil.getSession();
Transaction tx = null ;
try {
  tx = session.beginTransaction();←❷
  //faire votre travail←❸
  ...
  tx.commit();
}
catch (Exception e) {
  if (tx != null) {←❹
    try {
      tx.rollback();
    } catch (HibernateException he) {
      throw he;
    }
  }
  throw e;
}
finally {←❺
  try {
    session.close();
  } catch (HibernateException ex) {
    throw new XXXException(ex); //exception fatale
  }
}
```

Comme expliqué précédemment, la session s'obtient *via* `factory.openSession()` (repère ❶). La gestion de transaction (repère ❷) est indispensable, même s'il est possible de la rendre partiellement automatique et transparente, comme vous le verrez ultérieurement.

La plupart des appels aux méthodes de la session peuvent soulever une exception. Si cela intervient, il est primordial d'appeler un `rollback()` (repère ❹) sur votre transaction et d'ignorer la session en cours, son état pouvant être qualifié d'instable. Notez l'utilisation du bloc `finally` (repère ❺), qui garantit l'exécution de son contenu, que l'exception du bloc `try` précédent soit soulevée ou non.

Les actions de session

Vous allez maintenant vous intéresser de manière très générale aux actions que vous pouvez effectuer à partir d'une session. 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 instance persistante

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

Si vous connaissez son id, invoquez `session.get(Class clazz, Serializable id)` ou `session.load(Class clazz, Serializable id)`. En cas de non-existence de l'objet, `session.get()` renvoie `null`, et `session.load()` soulève une exception. Il est donc vivement conseillé d'utiliser la méthode `session.get()` :

```
Player player = (Player) session.get(Player.class, new Long(1));
// ou
// Player player = (Player) session.load(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=?
```

Ces deux méthodes permettent la récupération unitaire d'une entité persistante. Vous pouvez bien sûr former des requêtes orientées objet, *via* le langage HQL, par exemple.

D'Hibernate 2 à Hibernate 3

Notez l'abandon, dans Hibernate 3, de `session.find()` et `session.iterate()` au profit des API `Query`, `Criteria` et `SQLQuery`. Ces API et le langage HQL sont beaucoup plus performants et puissants qu'une gestion de requête par chaîne de caractères.

Rendre une nouvelle instance persistante

La méthode `session.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") ;
sess.persist(player);
```

```
insert into PLAYER (PLAYER_NAME, PLAYER_NUMBER, BIRTHDAY, HEIGHT, WEIGHT, TEAM_ID)
values (?, ?, ?, ?, ?, ?)
```

Il existe d'autres méthodes pour rendre une instance persistante, mais `session.persist()` a l'avantage de répondre à la spécification EJB 3.0. Dans le cas particulier des classes dont l'id est déclaré avec `unsaved-value="undefined"`, qui est le paramétrage par défaut lorsque le générateur `assigned` est choisi, utilisez la ligne suivante pour rendre une nouvelle instance persistante :

```
sess.merge(obj);
```

Elle effectue un select pour déterminer s'il faut insérer ou modifier les données.

Rendre persistantes les modifications d'une instance

Si l'instance persistante est présente dans la session, il n'y a rien à faire. Le simple fait de la modifier engendre une mise à jour lors du commit :

```
Player player = (Player) session.get(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, Hibernate, 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 une session, elle est dite *détachée.* Comprenez qu'il est impossible de traquer les modifications des instances persistantes qui ne sont pas attachées à une session. 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 en va de même pour tous les systèmes de persistance fondés sur les états des objets, comme TopLink ou encore JDO.

Pour réattacher à la session une instance détachée et rendre persistantes les modifications qui auraient pu survenir en dehors du scope de la session, il suffit d'invoquer la méthode `session.merge()` :

```
player = (Player)session.merge(player);
```

```
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=?
```

Le premier select est déclenché à l'invocation de `session.merge()`. Hibernate récupère les données et les fusionne avec les modifications apportées à l'instance détachée. `session.merge()` renvoie alors l'instance persistante, et celle-ci est attachée à la session.

Comme `session.persist()`, `session.merge()` répond aux spécifications EJB 3.0, mais il en existe plusieurs variantes.

Réattacher une instance détachée

Pour réattacher à une nouvelle session Hibernate une instance détachée, utilisez la méthode `session.lock()`. Elle diffère de `session.merge()` en ce qu'elle ignore les modifications apportées à l'instance avant son réattachement à une session. Si vous apportez des modifications à l'instance réattachée, celles-ci seront prises en compte au prochain `flush`.

Cette méthode permet aussi d'effectuer une vérification de version pour les objets versionnés ainsi que de verrouiller l'enregistrement en base de données.

`session.lock()` prend en second paramètre un `LockMode`, ou mode de verrouillage. Les différents modes de verrouillage sont récapitulés au tableau 2.7.

Le code suivant est à utiliser si vous récupérez une instance détachée que vous pensez modifier après réattachement à la session :

```
session.lock(player, LockMode.UPGRADE);
```

```
select PLAYER_ID from PLAYER where PLAYER_ID =? for update
```

Pour une gestion de concourance avancée, vous souhaiteriez vérifier la version afin d'être certain de travailler sur une instance « à jour ». De même, vous souhaiteriez mettre en place une file d'attente pour les accès concourants en déposant un verrou `for update`.

Si un accès concourant venait à s'effectuer, il serait placé en attente, l'attente se terminant une fois la transaction achevée.

Tableau 2.7. Modes de verrouillage

LockMode	*Select* pour vérification de version	Verrou (si supporté par la base)
NONE	NON	Aucun
READ	Select…	Aucun
UPGRADE	Select… for update	Si un accès concourant survient avant la fin de la transaction, il y a gestion de file d'attente.
UPGRADE_NOWAIT	Select… for update nowait	Si un accès concourant survient avant la fin de la transaction, une exception est soulevée.

Détacher une instance persistance

Détacher une instance signifie ne plus surveiller cette instance. Cela a 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 la session : `session.close()` ;
- en la vidant : `session.clear()` ;
- en détachant une instance particulière : `session.evict(obj)`.

Rendre un objet transient

Rendre un objet transient signifie l'extraire définitivement de la base de données. La méthode `session.delete()` 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 :

```
session.delete(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* `session.refresh()`.

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

> **La session Hibernate 2.1**
>
> Pour des raisons de compatibilité ascendante de vos applications, vous pouvez souhaiter utiliser l'ancienne API `Session` d'Hibernate 2.1. Pour cela, il vous suffit de l'importer depuis le package `org.hibernate.classic`.

Exercices

Pour chacun des exemples de code ci-dessous, définissez l'état de l'instance de `Player`, en supposant que la session est vide.

Énoncé 1:

```
public void test1(Player p){
←❶
  HibernateUtil.getSession();
  tx = session.beginTransaction();
  tx.commit();
  ←❷
}
```

Solution :

En ❶, l'instance provient d'une couche supérieure. Émettons l'hypothèse qu'elle est détachée. Une session est ensuite ouverte, 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
  HibernateUtil.getSession();
  tx = session.beginTransaction();
  Player p = session.get(Player.class,id);
  ←❶
  tx.commit();
  return p ; ←❷
}
```

Solution :

L'instance est récupérée *via* la session. 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){
  HibernateUtil.getSession();
  tx = session.beginTransaction();
  Player p = session.get(Player.class,id);
  ←❶
  tx.commit();
  session.close();
  return p.getTeam(); ; ←❷
}
```

Solution :

En ❶, l'instance est persistante, mais elle est détachée en ❷. La ligne ❷ 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
  HibernateUtil.getSession();
  tx = session.beginTransaction();
  ←❶
  session.delete(p);
  ←❷
  tx.commit();
  session.close();
}
```

Solution :

L'instance est détachée puis transiente.

Énoncé 5 :

```
public void test5(Player p){
  // nous supposons que p.getId() existe dans le datastore
  HibernateUtil.getSession();
  tx = session.beginTransaction();
  ←❶
  session.update(p);
  ←❷
  tx.commit();
  session.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 test6(Player p){
  HibernateUtil.getSession();
  tx = session.beginTransaction();
  ←❶
```

```
    session.lock(p, LockMode.NONE);
    ←❷
    tx.commit();
    session.close();
}
```

Solution :

Comme pour l'énoncé précédent, l'instance est détachée puis persistante. La différence est qu'ici les modifications effectuées sur l'instance lors du détachement ne seront pas rendues persistantes.

Énoncé 7 :

```
public void test7(Player p){
    HibernateUtil.getSession();
    tx = session.beginTransaction();
    ←❶
    Player p2 = session.merge(p);
    ←❷
    tx.commit();
    session.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 esquissé à la figure 2.5 avec les méthodes de la session Hibernate permettant la transition d'un état à un autre.

Conclusion

Vous disposez maintenant des informations nécessaires à la mise en place d'Hibernate en fonction de votre environnement de production cible (serveur d'applications ou base de données). 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 classes métier persistantes à votre schéma relationnel. C'est ce que nous vous proposons d'aborder au chapitre 3.

Figure 2.6

Cycle de vie des instances persistantes avec Hibernate