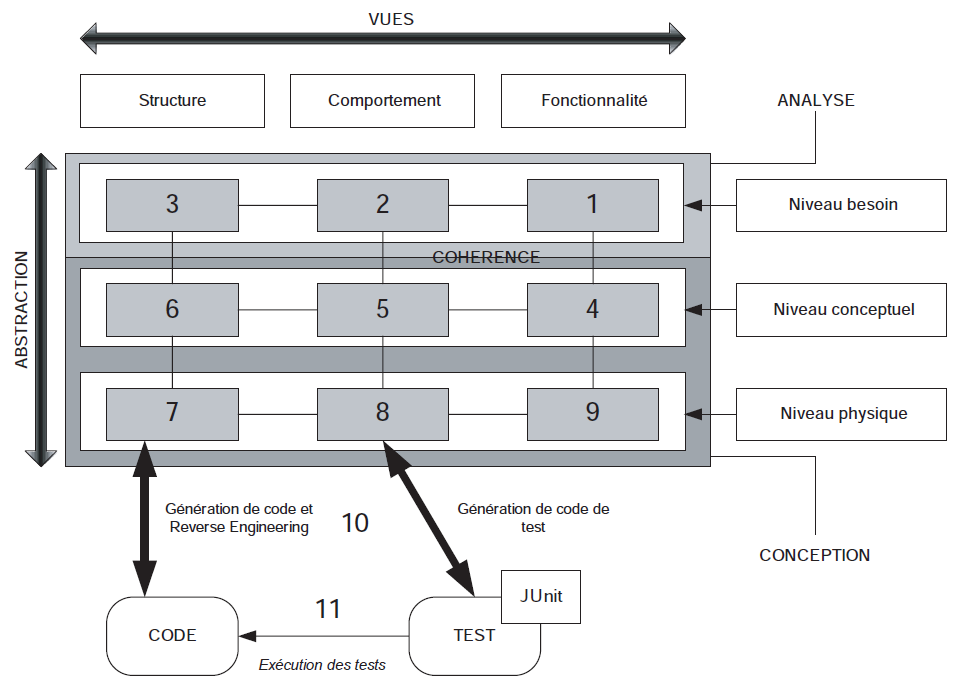
## UML Pour développeurs



Niveau besoin : correspond au recueil du besoin (cahier des charges + UML exprimant le besoin : use case et diagramme de séquence)

Niveau conceptuel : Diagrammes UML représentant la solution, et correspondant aux classes java développés

Niveau physique : Code dans la solution technique choisi.

Fonctionnalité : Diagramme de cas d’utilisation ( use case)  
Comportement : Diagramme de séquence   
Structure : Diagramme de classe.

Le niveau besoin et conceptuel ne sont pas reliés, mais il est possible de mettre des notes dans le conceptuel pour identifier les éléments correspondants au besoin résolus par ceux-ci.

Le niveau conceptuel et technique sont interconnecté en définissant des règles permettant aux outils UML de faire le reverse engineering et la génération de code. Cela permet au code de rester à jour sur le modèle, et vice versa. Le gain est de pouvoir voir un niveau au dessus pour appliquer des design pattern à l’uml, générer les tests Junit depuis le modèle, la documentation, et la correction des dépendances (éviter les dépendances croisés entre classes)

L’ordre de réalisation d’un projet partant de 0 va des étapes 1 à 9, à partir du cahier des charges.

Exemple de diagramme :

Diagramme de cas d’utilisation du système

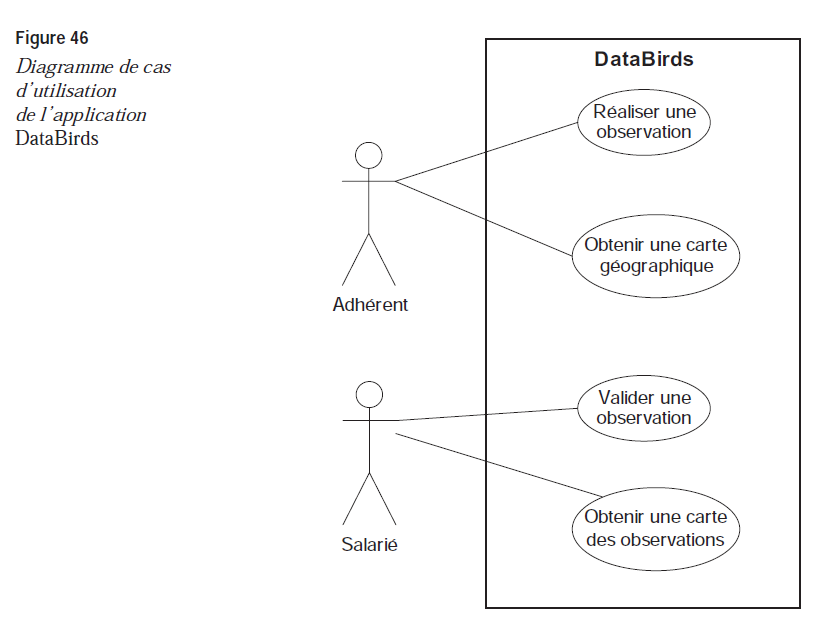


Diagramme de cas d’utilisation d’un composant du système

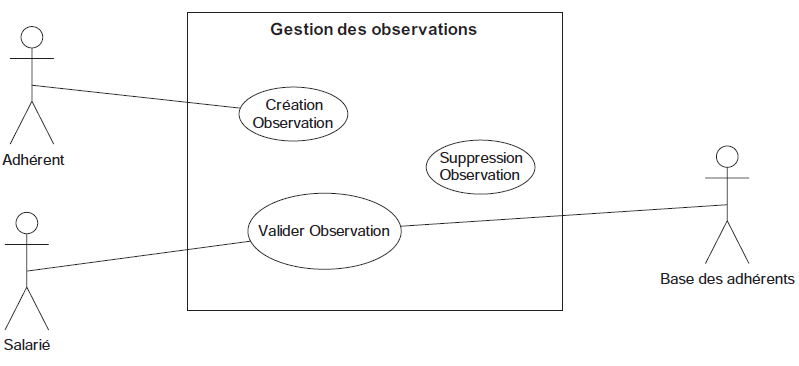


Diagramme de séquence d’un cas d’utilisation du composant

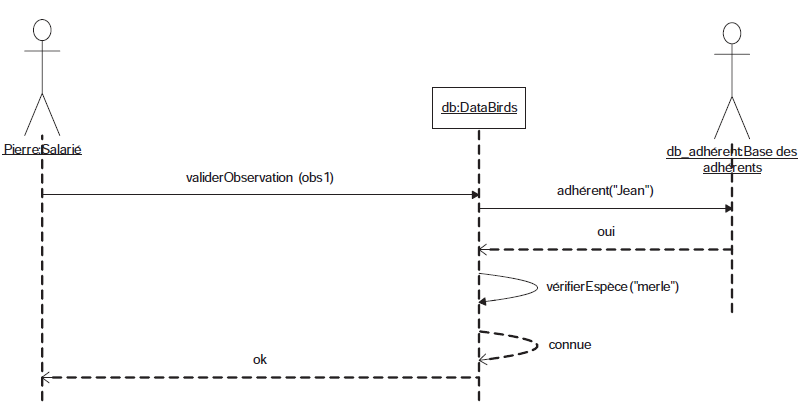
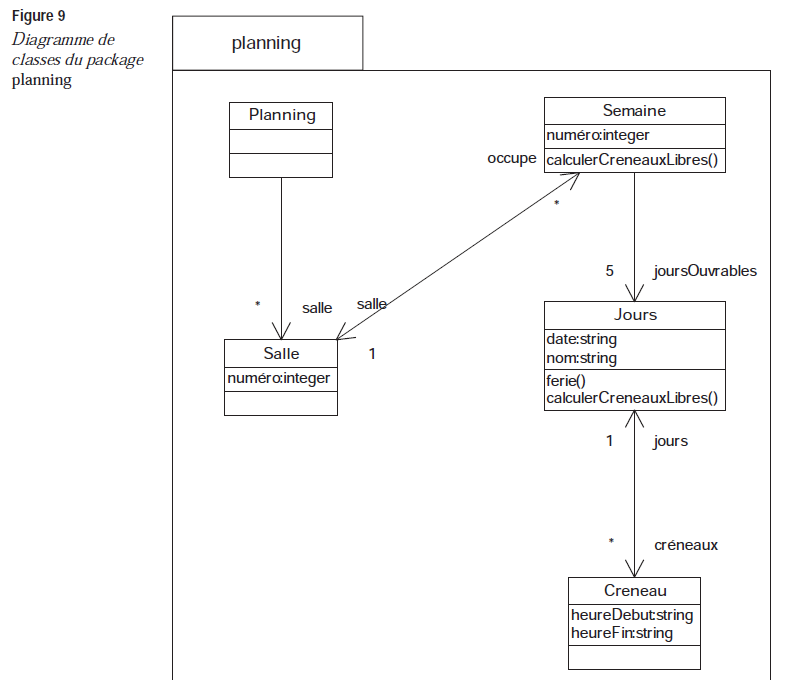
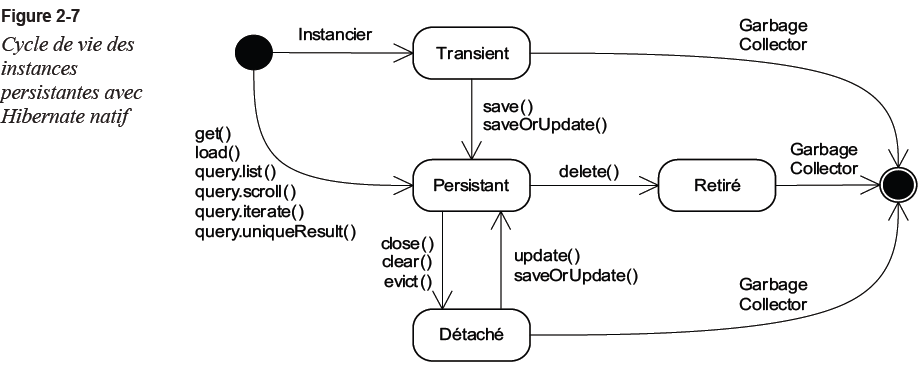


Diagramme de classe



## Hibernate



##### Annotation JPA ( Java Persistance) / Hibernate principale :

@Entity : Autour d’une classe, spécifie que cette classe est une entité, et donc amené à entre persistante.

@Table (optionnel): Spécifie la table sur laquelle l’entité est mappé.

@SecondaryTable – SecondaryTables : On peut vouloir mappé plusieurs tables à une entité, si cela n’a pas de sens particulier de créer deux entités.

@ID : Définie la clé primaire simple de la table ( simple= constitué d’une seule colonne). @GeneratedValue permet de générer automatiquement la valeur de la clé ( = sequence sous Oracle)

@IdClasse : en cas de clé composé, il faut regrouper ceux-ci dans une classe interne, et utiliser cet annotation.

@Embeddable : la classe est incluse dans une autre : Ex adresse dans Personne, mais on veut les stocker dans deux tables séparés en BD.

@Embedded : spécifie l’attribut de type classe qui est inclus. @EmbeddedId permet de spécifier que l’attribut en question est clé primaire, il remplace le @Id car il ne peut y en avoir qu’une.

@Basic & @Column (optionnel): Précise que l’attribut est mappé, et offre des options comme la longueur, l’unicité, la mise à jour ( fetch EAGER ou LAZY). Si on ne précise pas cet annotation sur un attribut, celui-ci est mappé avec les proprietés par défaut ( eager, nom de l’attribut = nom du champ en base)

@Transient : Précise que l’attribut n’est pas mappé, et donc n’est pas enregistré en base.

@Enumerated : pour les énum ( STRING et INT supportés)

@AttributesOverrides : précise que l’attribut en question surcharge les définitions d’objets inclus, mais aussi dans les mappings d’héritages.

##### Jointures :

@JoinColumn : permet de préciser que l’attibut est clé étrangère simpled’une autre table.  
@JoinColumns : dans le cas ou la clé étrangère est composite (constitué de plusieurs attributs. Ex :

@ManyToOne

@JoinColumns({

@JoinColumn(name="TEAM\_ID1", referencedColumnName="ID1"),

@JoinColumn(name=" TEAM\_ID ", referencedColumnName="ID2")

})

public Team getTeam() { return tem; }

@PrimaryKeyJoinColumn, @PrimaryKeyJoinColumns: table jointes par leur clé primaire: si on a une relation OneToOne ou dans la stratégie d’héritage joined, les tables doivent etre mappés par leurs clés primaires.

@ManyToOne et @OneToOne : en UML, il s’agit d’utiliser l’association simple, ou agrégation. Cela signifie que les instances des deux classes possèdent un cycle de vie indépendant ( contrairement à la composition). Chaque classe possède sa table.

L’’enumeration CascadeType de l’annotation permet de préciser les opérations en cascades de la classe agrégé : ALL, PERSIST, MERGE, REMOVE, REFRESH)

@ManyToOne : association UML de type \*--1.

@OneToOne: association UML de type 1--1. mappedBy (optionnel) doit etre précise dans le cas d’une association bi directionnelle : la classe agrégé pour savoir la proprieté de l’entité associé qui gère l’association.

##### Mapper les collections :

@OneToMany : unidirectionnel, pour gérer une collection (UML 1 🡪 \* ) ex :

@OneToMany(

cascade=CascadeType.ALL)

public Set<Player> getPlayers()

{ return players; }

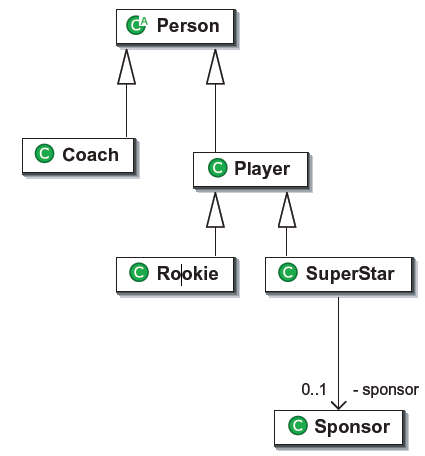
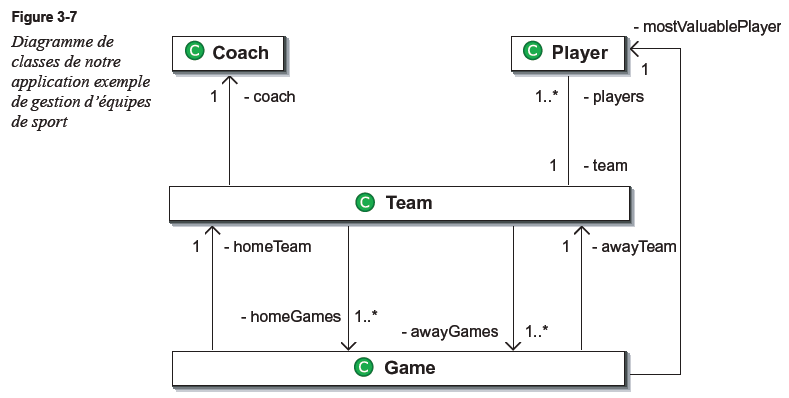
@ManyToMany : permet de définir une association de type UML \* < - - > \*. A toujours deux extrémités, dont une proprietaire de l’association ( possédant le @JoinTable) et l’autre contenant le nom de la proprieté proprietaire pour le mappage ( mappedBy)

@Order by : trie un élément de type classe dans une collection chargé par Hibernate. Celui-ci garantie l’ordre uniquement lors du chargement, et non par la suite.  
@MapKey permet de spécifier l’index si on choisit d’utiliser une MAP. . La propriété choisi doit avoir une contrainte d’unicité. Si on ne précise pas, c’est la clé primaire de la classe chargé qui est pris en compte par défaut

##### Heritage et Polymorphisme ( avancé)

Plusieurs stratégies existent pour mappés des problèmes complexes, tel que crée une table par classe concrète,

* une table par sous-classe ;
* une table par hiérarchie de classes ;
* une table par sous-classe avec discriminateur ;
* une table par classe concrète ;
* une table par classe concrète avec option « union ».



Stratégie : Une table par classe concrète : On utilise @MappedSuperClass pour la classe racine, mais pas @Entity car elle n’a pas de correspondance de table, les classes filles auront chacune leur table, on redéfinit au besoin les attributs et les asssociation en utilisant les annotations de surcharge (override).

@MappedSuperClass désigne une classe dont les informations de mapping s’appliquent aux entités qui héritent de la classe annoté. Ici on va mettre cette annotation sur la classe Personne, qui ne sera pas une entité relié à une table, mais ou l’on indiquera que certains de ces attributs, seront mappés dans les classes filles.

@AttributeOverride(s) va nous permettre dans la classe fille de surchargé le mapping d’attribut de la classe mère. L’annotation se place au niveau de la classe.

@AssociationOverride va nous permettre dans la classe fille de surcharger le mapping relatif à une associaiton ManyToOne ou OneToOne.

Stratégie : Héritage

Il existe 3 stratégies : TABLE\_PER\_CLASS, JOINED et SINGLE\_TABLE

@Inheritance est utilisé pour définir la stratégie d’héritage. A placer dans la classe racine, en association avec @Entity.

@Inheritance  
On garantie l’intégrité des données, mais pas le polymorphisme.  
 Exemple : on a une relation OneToMany pour récuperer un SET de personnes dans TEAM et une exception est levé car les identifiants des personnes de type COACH, PLAYER sont potentiellement identiques ( id unique par table mais pas entre 2 tables) .

Une table par classe concrète : TABLE\_PER\_CLASS   
@Inheritance(strategy =InheritanceType.TABLE\_PER\_CLASS)

Les identifiants des différentes tables sont générés par la même sequence. Cette méthode garantie le polymorphisme et l’intégrité des données, est efficace en insert (une table par instance ) mais est couteuse en lecture ( select des union des tables)

**Stratégie d’une table par classe concrète avec option union : JOINED**   
@Inheritance(strategy =InheritanceType.JOINED)  
Cette stratégie consiste à utiliser une table par sous classe , en plus de la table mappé à la classe mère : chaque classe est donc mappé à une table. Elle garantit le respect des contraintes not-null.  
Une création de COACH necessite une insertion dans PERSON et une autre dans COACH, car l’identifiant généré dans PERSON est réutilisé dans COACH. De plus, pour déterminer quel est la classe fille ( si on veut récuperer la personne dont l’id est 1 par exemple), la requete est obligé d’interroger toutes les tables entrant en jeu pour l’arbre d’héritage donné.  
Couteuse donc en lecture / ecriture, mais garantie intégrité des données et polymorphisme.

**Stratégie par discrimination SINGLE\_TABLE** :

Cette stratégie propose de mapper la hierarchie à une seule table.  
Une colonne discriminante, technique et transparente dans l’application est utilisé lors de la récuperation des données, pour décider quelle classe instancier.  
@DiscriminatorColumn défini la colonne (nom,type) dans la classe racine.  
Ex:   
@Entity(name="ch4.parHierarchie.Person")

@Table(name="HIERARCHIE\_PERSON")

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(

name = "PERSON\_TYPE",

discriminatorType = DiscriminatorType.STRING

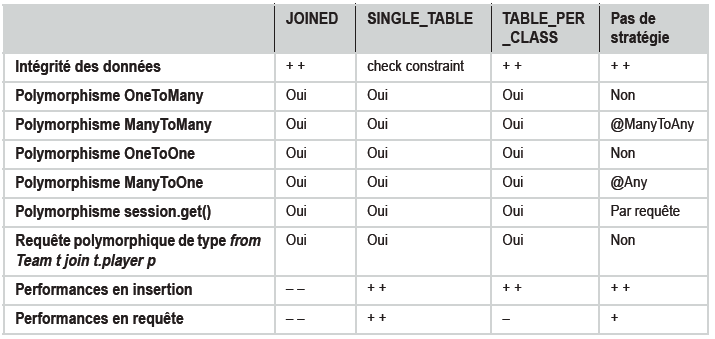
)

public abstract class Person

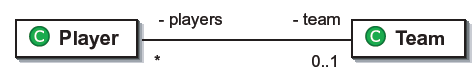
@DiscriminatorValue est utilisé pour définir la valeur dans la classe fille : ex @DiscriminatorValue("PLAYER").

Si des proprietés des classes filles ne peuvent être nulles, cette stratégie serait incapable de garantir l’intégrité des données du coté de la BD via des simples clauses not-null. Par exemple, si Rookie et Superstar hérite de Player, une Superstar a forcément un attribut SPONSOR\_ID , mais comme un Rookie n’en a pas, on ne peut pas mettre de contrainte « not null » en base de donnée ! Il faut donc utiliser une contrainte moins forte, de type check constraint.

Cette stratégie est excellente niveau performance, supporte le polymorphisme, mais pose en revanche problème quand aux contraintes à poser sur la BD, ce qui peut gêner dans certains projets.



Mise en œuvre d’une association bidirectionnelle



Coté métier, il faut créer une méthode métier « de cohérence » pour éviter d’écrire team.getPlayers().add(player) ET player.setTeam(team) (et éviter des oublis) , en l’occurrence il faut créer la méthode team.addPlayer(player) dans TEAM qui fera ces deux instructions.

Il est nécessaire d’indiquer à Java Persistence laquelle des deux extrémités de l’association

Bidirectionnelle est responsable de la gestion de la clé étrangère. Pour cela il faut utiliser mappedBy, disponible pour les annotations @OneToMany, @OneToOne et @ManyToMany.

@org.hibernate.annotations.CollectionOfElements permet de couvrir la fonctionnalité avancé de mapper ue collection de valeurs primitives ( int, String)

Dans une relation OneToOne, navigable dans les deux sens, il est necessaire d’utiliser mappedBy :

Dans Team:

@OneToOne

private Coach coach;

Dans coach :  
@OneToOne(mappedBy="coach")

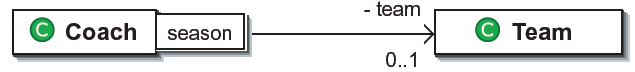
private Team team;

##### Collection d’objets inclus : stockage de l’information située dans les tables d’association

La table d’association devient une classe (composite)à part entière dans le MCD, et on utilisera l’annotation @CollectionOfElements dans la classe parent de la classe d’association et @Embeddable dans la classe composite. Son cycle de vie sera lié a l’instance Parent. On peut ajouter un pointeur depuis l’élément composite vers sont parent en utilisant l’annotation @Parent.

##### Relation ternaire : impliquant 3 classes

Exemple : Un coach entraine une équipe, pendant une saison défini par une date de début / une date de fin.



Soit Season est fortement dépendant de COACH auquel il appartient, et sera donc un objet inclus de coach, soit il possède son propre cycle de vie : c’est une entité.

Objet inclus : on utilise @Embeddable dans COACH pour annoter Season, puis @MapKey pour mapper les équipes / saison de cette façon :  
@ManyToMany  
@MapKey  
private Map<Season,Team> teams = new HashMap<Season,Team>();

Grace à l’utilisation de la MAP, les mappings liés à la classe Season seront résolus automatiquement par Hibernate. @MapKey est utilisable uniquement avec comme clé des objets inclus.

Relation ternaire avec entité :  
L’utilisation est la même, mais on utilise l’annotation @MapKeyManyToMany

##### Association n-aire :

Il faut mixer le mapping du qualifier entité, et celui de la collection d’objet inclus :

@CollectionOfElements

@MapKeyManyToMany

**private** Map<Season,CoachTeam> coachTeams = **new** HashMap<Season,CoachTeam>();

Penser à ajouter les methods métier pour gérer cette association dans son intégralité depuis une méthode centralisé:  
public void createCoachTeam(Season season, Person director, Team team){

CoachTeam coachTeam = new CoachTeam();

coachTeam.setCoach(this);

coachTeam.setDirector(director);

coachTeam.setTeam(team);

this.getCoachTeams().put(season, coachTeam);

}

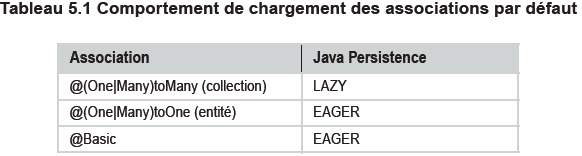
L’association ManyToMany de base est rare, car elle n’apporte pas de valeur ajouté, en lieu et place on rencontrera plutôt des associations n aire et des cas nécessitant l’emploi d’élements composites.

##### Méthode de récuperation des données

Hibernate fournit une solution complète pour maitriser l’accès aux données de façon optimisé par :

* configuration du chargement a la demande, ou lazy loading, via annotation afin d’éviter le chargement d’instances inutiles, en opposition avec l’eager fetching.
* chargement des instances associées à l’exécution, pour charger un réseau d’objet plus large et limiter le nombre de requêtes générés.
* optimisation du SQL généré, prenant en compte les spécificités du dialecte de la BD cible.

Par défaut, les collections ( One-ManyToMany) ne sont pas chargées lors de la récupération de l’entité à laquelle elles sont associées. Le set est simplement initialisé et sa session crée ( attribut spécifique d’hibernate « PersistentSet »). Ce comportement est modifiable via annotation et même lors de l’execution.  
Les relations (One-ManyToOne) sont en revanche chargé de suite, car en général cela n’a pas d’impact sur les performances, la jointure externe ne multipliant pas le nombre d’enregistrement retournés.



Paramétrage du chargement via fetch( charger en anglais)  
Ce membre est disponible dans les annotations lié à la déclaration d’association et éléments liés.  
FetchType.LAZY ou FetchType.EAGER. Par exemple pour charger à la demande l’équipe :  
@ManyToOne(fetch=FetchType.LAZY)

private Team team;

Hibernate utilise intensivement les proxy .Proxy est un design pattern qui comporte de nombreuses variantes. Hibernate l’utilise en

tant que déclencheur pour gérer l’accès à une entité. Si le proxy est accédé, il a la faculté de se remplacer lui-même par la véritable entité par les opérations :

* Lecture dans le gestionnaire d’entités.
* Lecture dans le cache de second niveau, si celui-ci est configuré et si le comportement

par défaut n’est pas surchargé.

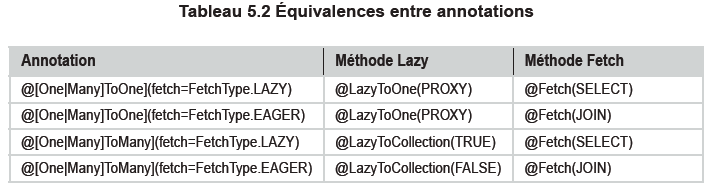
* Interrogation en base de données, si cela s’avère nécessaire, comme dans notre test

Pour une gestion plus fine, Hibernante propose les annotations :

@LazyToOne, pour définir la méthode à utiliser pour effectuer le lazy loading vers les entités ToOne : PROXY : par défaut, utilise la génération de proxy dynamique  
No\_PROXY : instrumentation du bytecode, via une tache Ant dispo dans Hibernate.  
FALSE : l’association ne peut pas être lazy.

@LazyCollection : sur les collections annoté en ToMany. Les options sont :   
TRUE : par défaut, collection chargé lorsqu’on y accède  
EXTRA : évite au maximum le chargement de la collection, par exemple si on ne veut savoir que sa taille.  
FALSE : l’association ne peut pas être lazy

@Fetch : définit la méthode à utiliser lorsque le chargement intervient. 3 paramètres :  
SELECT : un select SQL est généré lors de l’accès à l’association  
SUBSELECT : pour les collections seulement, permet de charger en une requetes toutes les collections ayant le même role. Ex : si on charge om et psg , et que l’on accède à om.players, cela charge en même temps psg.players  
JOIN : inhibe le paramètrage lazy, la collection est chargé au moment du chargement de l’entité racine.



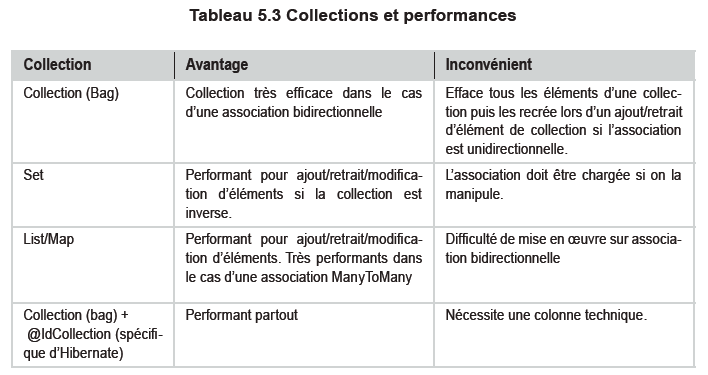
Selon le type de la collection retenue pour un mapping particulier, un impact peut être observé

sur le chargement des associations. En lecture, tous les types de collections adoptent le lazy

loading, tandis qu’en écriture leur comportement diffère d’un type à un autre.  
Set et List oblige le chargement entier de la collection en cas d’ajout, Set pour vérifier que la valeur n’est pas présente, List pour récuperer l’index de l’élément à inserer et respecter l’ordre.  
La Collection de base ne necessite pas de requetes de chargement, pour ajouter un élément à celle-ci, mais en revanche pose problème lors du persist en BD, puisque pour respecter la contrainte d’unicité, tout les élements vont être effacé et recréer un par un pour éliminer les doublons éventuels ( pas de solution miracle donc)

@CollectionId permet de parer au problème de la collection standard. Les élements de la collection sont ainsi identifiés et indexés :  
@OneToMany(cascade = CascadeType.PERSIST)

@org.hibernate.annotations.CollectionId(columns = @Column(name="COLLECTION\_ID"),type=@org.hibernate.annotations.Type(type="long"),generator="ids\_generator")



L’experience montre que la plupart des modélisations adoptent le Set et que les associations sont principalement bidirectionnelles.

##### Technique de récupération d’objets – Langage HQL

Em.find() et em.getReference(), pour récuperer une entité par son identifiant.

EJB-QL, via Java Persistence

HQL , spécifique d’hibernate, offrant le support d’EJB-QL  
SQL natif  
API Criteria, spécifique d’Hibernate, par une construction programmatique des requêtes.

StringBuffer queryString = new StringBuffer();

queryString.append("requête en EJB-QL")

Query query = em.createQuery(queryString.toString());

List results = query.getResultList() ;

OU

Object o = query.getSingleResult() ;

La méthode em.createQuery(requete) prend en argument une chaîne de caractères. Une bonne

pratique consiste à utiliser systématiquement la classe StringBuffer pour la concaténation des parties constituant la chaîne. L’opérateur + appliqué à la classe String demande plus de ressources que la méthode append().

La requete interroge une classe et non une table.

Le langage étant fondé sur le raisonnement objet, la navigation dans le réseau d’objets apporte une simplicité d’écriture et de lisibilité :

Ex: “Select team.coach from Team team” ou “Select player.team.coach from Player p”

Les elements d’une collection peuvent être retournée par une requites grace au mot clé elements:  
Select elements(team.players) from Team team.

Le polymorphisme est nativement supporté par les requêtes. Si on interroge la classe abstraite Person, le résultats seront des entités de classe fille Coach, Player etc.

Jointure normale : select team from Team team **join** team.coach (retourne les équipes ayant un coach)  
Jointure ouverte : select team from Team team **left join** team.coach (retourne les équipes ayant un coach et celles qui n’en ont pas)

En cas de jointures, les requêtes peuvent retourner plusieurs fois les mêmes instances. Il faudra soit utiliser **distinct**, soit utiliser le **Set** qui dédoublonnera en mémoire : Set distinctResults = new HashSet(results) ;

Clause where : identique à SQL

Passage de paramètres à une requete : pour éviter le trou de sécurité de passer des paramètres par des String ou StringBuffer venant d’interface utilisateur et donc potentiellement dangereux, il faut utiliser le caractère ? ( paramètre positionnés) .   
queryString.append("select team from Team team where team.name = :name");  
query.setParameter("name",param);

Plus économe en ligne, on peut aussi passer en parameters des beans à condition que le nom des attributs soit proche de ceux de l’entité.

query.setProperties(dto);

**@NamedQuery**  permet de déclarer des requetes sous forme d’annotation. La requète peut être récupéré dans le code de la façon suivante : Query query = em.createNamedQuery(« myNamedQuery ») ;

##### Chargement des associations

Principe : on peut charger autant d’entités désiré ( ToOne), il faut éviter de charger plusieurs collections, au maximum une pour éviter de faire des produits cartesiens de tables. Pour charger deux collections, il est plus sur d’exécuter deux requêtes à la suite.

Rêgle importante : Il est impossible d’utiliser une collection chargée par requête dans une restriction !

Pour charger explicitement les collections lié à une entité, on peut utiliser join fetch :

queryString.append("select team from Team team ")

.append("**left join fetch team.coach** c ")

.append("**left join fetch team.players** p ");

L’aspect objet d’HBQL permet de réduire le nombre de ligne par rapport au SQL.

Il reste un problème, lorsque les éléments d’une collection possèdent eux même une association vers une tierce entité, celle-ci n’est pas chargé avec la collection, et il faut parcourir un à un les éléments de la collection pour la charger ( n+1 requetes necessaire, n étant le nombre d’élements de la collection).

Il existe deux solutions à ce problème

##### Charger plusieurs collections

Dans la plupart des cas, il est interessant de charger **en découpant plusieurs requetes**, plutôt qu’en une grosse, ou en n+1 requetes.  
Le **chargement par lot** améliore aussi les performances : batch fetching en annotant la collection par **@BatchSize(size=n)**. Ex :  
@org.hibernate.annotations.BatchSize(size=3)

private Set<Characteristic> characteristics =

new HashSet<Characteristic>(); -> les collections characteristics sont chargés par paquetes de 3 à la demande. On réduit donc le nombre de requêtes par 3 !!  
Au programmeur de décider le nombre à appliquer, celui-ci étant plus ou moins facile à optimiser : il faut faire des estimations / stats, ou bien utiliser @Fetch(SUBSELECT)

On peut aussi utiliser **@BatchSize** sur une entité, et bénéficier des mêmes avantages dans le cas de relation ToOne.

Notion de verrou

Verrou pessimiste :

Verrou en ecriture : bloque l’accès a la donnée aux threads concurrents que ce soit en lecture / ecriture / suppression

Verrou en lecture : indique aux autres thread que la donnée est en train d’être lu. Ils ne peuvent donc y accéder qu’en lecture seule.

Les verrous sont peu utilisés car ils dégradent les performances en environnement concurrent.

Verrou optimiste :

On va poser un verrou en lecture au moment de la lecture des données, et un verrou en écriture au moment de l’écriture. Si entre temps un thread concurrent a touché a la donnée, une collision est possible, il est question de gérer et résoudre les collisions.

La gestion optimiste avec versionnement est la solution ayant le meilleur rapport cout / performance.

La table cible contient une colonne technique indiquant la version. Lors de la mise à jour, est ajouté automatiquement une clause where avec un prédicat sur le numéro de version préalablement lu.

Pour ajouter la gestion avec versionnement, il suffit d’ajouter

@Version

private int version;

dans l’entité.

Hibernate permet d’aller plus loin sans création de colonne technique en utilisant

optimisticLock=org.hibernate.annotations.OptimisticLockType.DIRTY)

-> le champ modifié sera vérifié au moment de l’update

ou

optimisticLock=org.hibernate.annotations.OptimisticLockType.ALL)

Dans ce cas tout les champs sont vérifiés, on peut en exclure via l’annotation @org.hibernate.annotations.OptimisticLock(excluded=true)

En mode Web/autonome (webapp à base de Servlet et

Tomcat typiquement), il n’est pas concevable de laisser une connexion JDBC ouverte

entre deux actions de l’utilisateur sur deux pages Web différentes.

Page 241

##### Obtenir et manipuler le gestionnaire d’entité

le gestionnaire d’entités n’est pas un cache global. Si deux traitements,

ou threads, parallèles venaient à utiliser un même gestionnaire d’entités, Java

Persistence ne pourrait garantir les données.

L’EntityManagerFactory dispense les gestionnaires d’entités. Le gestionnaire d’entité surveille les entités tant que le contexte de persistance existe, c'est-à-dire pendant le contexte de temps court nécessaire au traitement, dit « unité de travail ».

Il existe deux types de contextes de persistance : le contexte de persistance porté par la transaction et le contexte de persistance étendu

Par la transaction :   
le cycle de vie de persistance commence au début de la transaction et se termine quand celle-ci s’acheve.

Etendu :

Le contexte de persistance continue sur plusieurs transactions. Il est utile pour gérer une conversation, par exemple une suite de formulaire web que l’on souhaite valider à la dernière page.

En environnement autonome ( jboss non integré), il nous faut gérer la connexion au datasource et la création du gestionnaire d’entité. La connexion au datasource se fait dans le fichier xml META-INF/persistence.xml. La création du gestionnaire d’entité se fera via une classe utilitaire ayant une variable statique EntityManagerFactory.

Pour la transaction, on l’obtient via l’interface EntityTransaction

EntityTransaction tx = em.getTransaction();

Pour la gestion des exceptions, il faut veiller à fermer l’Entity Manager dans tout les cas, grace à un try catch finally :

Try{

…  
em.persist(team);

tx.commit();

} catch (RuntimeException ex) {

try {

tx.rollback();

} catch (RuntimeException rbEx) {

}

throw ex;

} finally {

em.close();

}

##### Conversation

Une conversation est un état nécessitant d’être maintenue pour enregistrer les modifications d’une série de page.

La synchronisation entre le gestionnaire d’entité et la base de donnée s’appelle le flush, il est possible de parametré le moment ou celui-ci est effectué

* flushMode.COMMIT. Le flush est effectué au commit de la transaction.
* flushMode.AUTO (défaut). Le flush est effectué au commit et avant l’exécution de certaines requêtes afin de garantir la validité du résultat.
* flushMode.ALWAYS. La synchronisation se fait avant chaque exécution de requête. Ce mode pouvant affecter les performances, il est déconseillé de changer le FlushMode sans raison valable. Seul le flushMode.AUTO garantit de ne pas récupérer dans les résultats de requête des données obsolètes par rapport à l’état de la session.
* flushMode.MANUAL. La base de données n’est pas automatiquement synchronisée avec la session Hibernate. Pour la synchroniser, il faut appeler explicitement session.flush().

En mode auto, la mise à jour peut etre appelé avant la fin de la transaction, si on modifie par exemple un joueur, puis que l’on fait un select sur la table des joueurs, le insert va être exectué avant le select, pour garantir que celui-ci soit exact.

Prenons le cas d’une conversation se déroulant sur plusieurs étapes, comment garantir au mieux que les données modifiés à chaque étape seront enregistrés en base de données ?

###### Gestionnaires d’entités multiples et objets détachés

1er solution pour gérer une conversation : chaque étape aura son gestionnaire d’entité et sa transaction, les objets sont chargés et détachés à chaque étape. L’utilisation d’entités détachées est délicate même si hibernate propose des méthodes pour réattacher des entités détachés avec session.merge et session.lock. La solution de charger tout le graphe d’objet au début de la conversation et de faire passer les modifications par un merge se propageant a la fin est tentante mais peut être pénalisante en terme de performances, de même que ne pas charger assez d’entité peut aboutir à des LazyXXXException difficile à résoudre.

###### Mise en place d’un contexte de persistance étendu

Consiste à garder le gestionnaire d’entité en vie tout le long des différentes étapes. Chaque étape a sa transaction. Il faut éviter tout flush avant la validation de la conversation au dernier écran.-> Stocker le gestionnaire d’entité dans la session , et passé en mode flushMode.MANUAL pour gérer le flush uniquemetn a la fin.

@Stateful

public class ConversationDemoExtended implements ConversationDemo {

@PersistenceContext(

unitName = "eyrollesEntityManager",

type = PersistenceContextType.EXTENDED,

properties = @PersistenceProperty(

name="org.hibernate.flushMode",

value="MANUAL")

)

private EntityManager em;

Lors du flush, il faut retirer l’instance de l’EJB Session du conteneur. Cela se fait par l’annotation  
@Remove

public void validateModification() {

em.flush();

}

###### Manipulation du gestionnaire d’entité dans un batch

Java n’est pas le language le plus adapté pour les traitements de masse, cependant a condition de bien coder, cela se fait avec un surcout négligeable en terme de performance. Il faut veiller à configurer la taille de notre gestionnaire d’entité dans persistence.xml :   
<property name="hibernate.batch\_size" value="20"/>

Ainsi dans le cadre d’un batch, le gestionnaire d’entité se videra tout les 20 enregistrements, évitant un OutOfMemoryException. Il faut donc appelé flush tout les 20 enregistrements.

On peut également optimiser la génération d’identifiant.  
Scroll : Permet d’optimiser la consommation mémoire / nb de requete executé :

ScrollableResults teams = session.getNamedQuery("GetTeams")

.scroll(ScrollMode.FORWARD\_ONLY);

int count=0;

while ( teams.next() ) {

Team team = (Team) teams.get(0);

team.updateStuff(...);

if ( ++count % 20 == 0 ) {

//synchronise la base de données avec les mises

//à jour et libère la mémoire

session.flush();

session.clear();

}

}

tm.commit();

Requete EJB-QL : il est possible d’executer seulement des requetes SQL sans transformation en entité, ni cache. Attention car le cache n’est alors plus syncrhonisés

tm.begin();

Query q = em.createQuery("update Team t set t.name = :param");

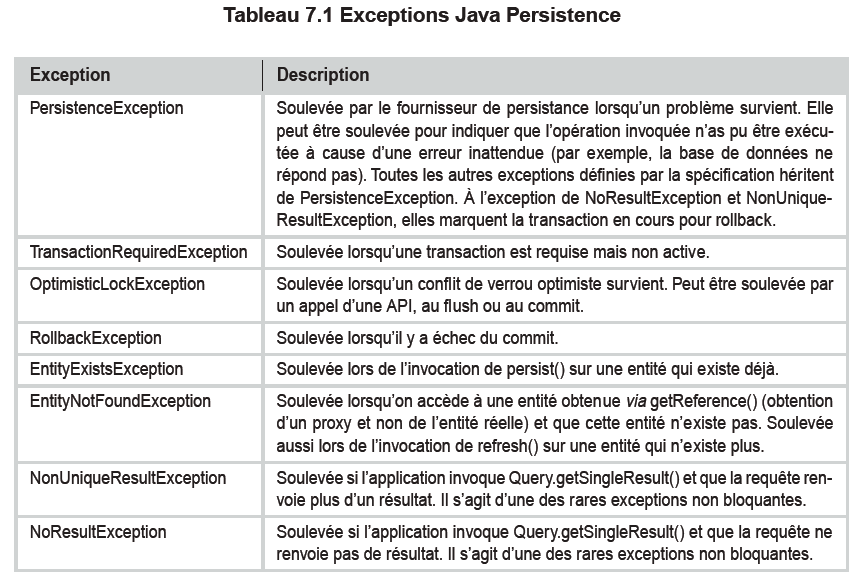
q.setParameter("param", "blah");

q.executeUpdate();

tm.commit();

250

##### Gestion des exceptions



##### Fonctionnalités avancés d’hibernate

###### Gerer des clés composés

Pour gérer des clés composés, il convient de créer une classe destiné à stocker la clé. Celle-ci doit comporter une constructeur public et être marqué comme @Serializable, et implémenter equals() et hashcode().

3 possibiltés existent :  
Marquer la classe de la clé composé grace à @Embeddable, et marqué celui-ci dans la classe de l’entité comportant la clé composé avec @Id.  
**Ne pas stéréotypé la clé composé, mais juste l’inclure en utilisant @EmbeddedId.**  
Ne pas stéréotypé la clé composé, copier les proprietés de la clé composé dans la classe, et annoté @Idclass la proprieté Id. Déconseillé car lourd à implémenter

Pour récupérer une entité en faisant une recherche par sa clé composite, vous pouvez utiliser em.getReference() et em.find(), en instanciant la classe composite et en la passant en paramètre.

Clé étrangère ne référençant pas la clé primaire

Ce cas peut arriver et est gérer par Hibernate, en déclarant soit une proprieté comme étant unique, garantissant la relation OneToOne, soit par un ensemble de proprietés uniques.  
1 proprieté : Annoter la colonne avec   
@Column(name = "UNIQUE\_PROPERTY", nullable = false, **unique=true**)

private int uniqueProperty;

puis dans la classe à relier, de façon indiquer la clé étrangère :  
@OneToOne

@JoinColumn(name="COACH\_REFERENCE",

**referencedColumnName = "UNIQUE\_PROPERTY")**

private Coach coach;

Plusieurs proprietés :   
La classe à relier sera annoter avec la liste des clés étrangère :

@OneToOne

**@JoinColumns({**

**@JoinColumn(name="COACH\_REFERENCE\_1",**

**referencedColumnName = "UNIQUE\_PROPERTY\_1"),260**

**@JoinColumn(name="COACH\_REFERENCE\_2",**

**referencedColumnName = "UNIQUE\_PROPERTY\_2")**

})

private Coach coach;

###### Les formules :

Si l’on souhaite calculer combien de fois un joueur a été élu meilleur joueur du match, on peut imaginer deux appels à la couche persistance, un pour charger le joueur, un deuxième pour calculer le nombre de fois ou il a été élu meilleur joueur. Afin de rendre la classe plus efficiente, la solution la plus élégante est **la création d’une proprieté hasBeenMvpCount avec une formule**. On ne fait donc plus **qu’un seul appel à la couche de persistance** :  
@org.hibernate.annotations.Formula(

"select count(\*) from GAME g where g.MVP\_ID = ID")

private int hasBeenMvpCount;  
Attention, cela génère une requete SQL supplémentaire lors du chargement de Player. Il faut donc l’associer avec du lazy loading.

**Les formules peuvent aussi servir pour discriminer une table**. Si on utilise l’héritage, avec la classe smallplayer et bigplayer et la stratégie SINGLE\_TABLE, la formule peut être placé sur la table player ainsi :

**@org.hibernate.annotations.DiscriminatorFormula(**

**"case " +**

**"when height is null then 'player' " +**

**"when height > 2 then 'bigplayer' " +**

**"else 'smallPlayer' end"**

**)**

@DiscriminatorValue("player")  
public class Player extends Person{…}

###### Les triggers et cascade On Delete

Un trigger est opposé au concept d’hibernate, car il modifie la base de données sans que le gestionnaire d’entité en est conscience, et corrompt les données . Le gestionnaire d’entité doit donc rafraichir son état après execution d’un trigger.  
L’annotation @org.hibernate.annotations.Generated placée sur un attribut permet de résoudre cela , prend en paramètre :  
org.hibernate.annotations.GenerationTime.NEVER, INSERT (insertion), ALWAYS (insertion et modification). Il faut positionner insertable=false, updatable=false pour le cas ALWAYS pour que cela fonctionne.  
Exemple :  
@Column(insertable = false, updatable = false)

@org.hibernate.annotations.Generated(

org.hibernate.annotations.GenerationTime.ALWAYS

)

private Date dateModification;

@org.hibernate.annotations.OnDelete placer sur une association permet de gérer le cas ou c’est la base de données qui va supprimer les enregistrements en cascade via un trigger et non le gestionnaire d’entité. Il prend en paramètre :  
org.hibernate.annotations.OnDeleteAction.NO\_ACTION : ne fait rien , option par défaut  
org.hibernate.annotations.OnDeleteAction.CASCADE : stipule que la BD prend en charge l’effacement des enregistrements correspondants à l’association annotée. Ex :  
@Entity

public class Player {

…

@ManyToOne

**@OnDelete(action=OnDeleteAction.CASCADE)**

public Team getTeam() { … }

}

###### Ordres SQL et procédures stockées

Hibernate se charge de générer les requetes SQL adapté à la base de donnée connecté. Il est cependant possible de « forcer » les requêtes à utiliser grâce aux annotations :

**@org.hibernate.annotations.Loader** : définit la requête à utiliser pour charger une entité.

**@org.hibernate.annotations.SQLInsert**: personnalisation de l’insert

**@org.hibernate.annotations.SQLUpdate**: personnalisation de l’update

**@org.hibernate.annotations.SQLDelete**: personnalisation du delete

**@org.hibernate.annotations.SQLDeleteAll :**

Insert, Update & Delete prenne 3 paramètre : sql ( requete), callable : true si il s’agit d’un procédure stockée, check : style de vérification adopte ( none, count ( = rowcount), param = retour personalisable)

Ces annotations peuvent être placé au niveau de l’entité mais aussi au niveau des tables secondaires avec @org.hibernate.annotations.Table

###### Gestion des clés étrangères corrompues

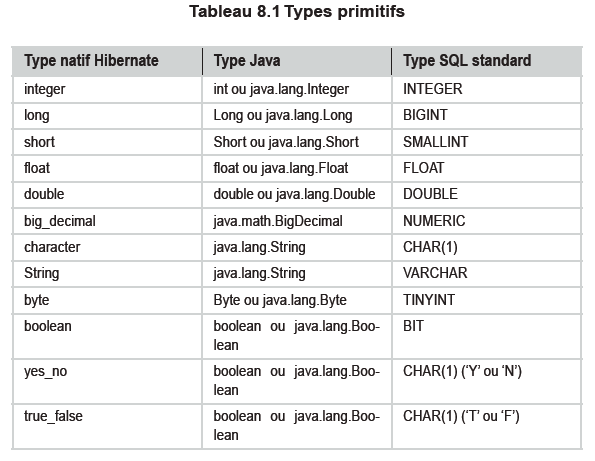
Parfois la **désactivation des contraintes sur les clés étrangère** est appliquée pour de mauvaises raisons. Le pire cas est la suppression de la clé étrangère. Si le TEAM\_ID d’un player ne correspond à aucune team, player.getTeam() ne pourra être résolu.  
**@org.hibernate.annotations.NotFound** défini le comportement à adopté dans ce cas : Exception lève une exception, IGNORE : ignoré que l’élément ne se trouve pas en base de données

Chargement tardif des proprietés

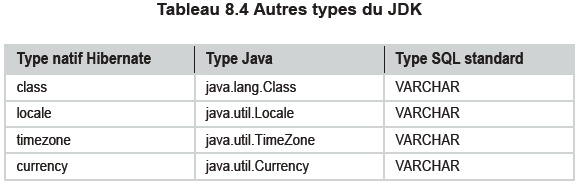
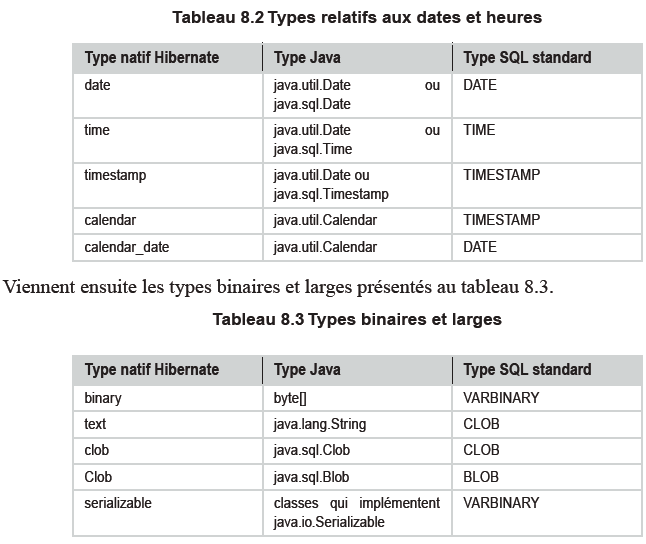
###### Chargement tardif des proprietés

Le lazy loading des associations est également disponible pour les proprietés -> le chargement est adapté pour **les longs texte**, ou les **classes ayant beaucoup de proprietés** à charger, et **les proprietés calculés via des formules**. L’annotation a utilisé est :  
**@Basic(fetch=FetchType.LAZY)**

**Typage**



Si le JDBC gère les type clob et blob, ils sont chargés à la demande par celui-ci.



###### Les types personnalisés ( User Type)

Il arrive des cas ou la proprietés en base et l’attribut de la classe semble impossible à mapper. Exemple : un score contenu dans un varchar en base ( 2 – 1) et en représentation objet   
public class Score {

private int homeTeamScore;

private int awayTeamScore;

…

}  
les User Type permettent de résoudre cette problématique. Il faut que la classe Score implémente l’interface **UserType** ( il existe CompositeUserType, UserCollectionType, EhancedUserType, UserVersionType, ParametizedType). Pour l’utiliser dans l’entité Game :  
**@org.hibernate.annotations.Type(**

**type = "ch8.demoUserType.ScoreUserType")**

@Column(name = "SCORE")

private Score score;

Les méthodes de l’interface à implémenter sont  
sqlTypes() : indique le type de colonne à utiliser pour la génération du schéma vie SchemaExport  
returnedClass() : indique la classe exploitée, ici Score  
isMutable() : indique si les instances sont muables ou non  
deepCopy() : utilisé par le gestionnaire d’entité pour le dirty checking ( vérifie si il y a eu des changements). Doit retourner une copie de l’objet  
dissassemble() : sérialisation des informations de l’objet (utilisé par le cache Hibernate)  
assemble() : instanciation de l’objet depuis un flux (utilisé par le cache Hibernate)  
replace() : retourne une copie profonde de l’objet, utilisé lors du merging em.merge()  
equals() et hashCode()  
nullSageGet() : transcode la colonne de la BD -> l’attribut  
nullSageSet() : transcode l’attribut -> la colonne de la BD

###### Filtres et interception d’evenement

**Les filtres sont utilisés principalement pour l’internationalisation et la sécurité des applications.**

**Les filtres de collection** sont des requetes non pas effectués en base mais sur une collection. Session. Createfilter() prend en paramètre la collection à filtrer et une requete EJB-QL et renvoie une liste d’entités.  
Collection smallPlayers = **session.createFilter**(t.getPlayers(),"where this.height < :height ").setParameter( "height", new Float(1.88)).list();

**Filtre statique sur une entité** via @org.hibernate.annotations, permet d’appliquer une clause where de façon automatique  
@OneToMany(mappedBy="team", cascade=CascadeType.MERGE)

**@org.hibernate.annotations.Where(clause="name is not null")**

private Collection<Player> players = new ArrayList<Player>();

**Filtre dynamique sur une entité**  
2 étapes : définir le filtre et l’appliquer  
@org.hibernate.annotations.FilterDef permet de définir le filtre , son nom et les paramètres dynamiques  
@org.hibernate.annotations.Filter prend en paramètre le filterdef et la condition.

Il faut ensuite appliquer le filtre en lui donnant ses paramètres dans le code :  
Filter filter = session.enableFilter("limitPlayerByHeight");

filter.setParameter("height", new Float(1.88));

Le filter s’applique de la même façon sur une collection.

**Filtre dynamique sur une association** :  
**@org.hibernate.annotations.FilterJoinTable** s’utilise de la même façon.

Le mélange de filtre dynamique ne pose pas de problème.

###### Interception d’evenement

Un evenement peut être assimilé à une action menée sur le cycle de vie d’une entité. Les callbacks sont un moyen d’intercepter chaque événement du cycle de vie de l’entité :  
• PrePersist

• PostPersist

• PreRemove

• PostRemove

• PreUpdate

• PostUpdate

• PostLoad

Il existe deux moyens de déclencher des traitements particuliers :   
Dans l’entité même via les annotations . 1 méthode par type d’annotation seulement!

@PrePersist

protected void notifyPersistenceRequest(){

System.out.println(this.getName()

+ " sur le point d'être rendu persistant");

}

Il est aussi possible d’externaliser ces méthodes dans une classe Listener pouvant prendre en charge les callbacks de plusieurs entités. La classe entité doit avoir l’annotation @EntityListeners(Monitor.class) , monitor étant la classe listener.

Dans la classe Monitor, chaque méthode doit prendre en paramètre l’entité écoutée.

On peut mixer les deux méthodes, l’ordre d’execution sera d’abord le listener puis les fonctions de classe entité.

###### Mapping modèle dynamique/relationnel

Afin de répondre à des besoins exotique, Hibernate propose des mappings de classes dynamiques et le mapping XML / Relationnel, ils se font via le fichier de mapping XML.

Mapping de classe dynamique

Exemple : Map de Map. L’utilisation de classes dynamique dispense de coder des classes persistante, et n’utilise plus l’entité « name » de l’élément « class » mais « class entity-name= Coach>

<hibernate-mapping >

<class **entity-name**="Team" table="TEAM" >

<id name="id" column="TEAM\_ID" type="long">

<generator class="native"/>

</id>

<many-to-one name="coach" **column="COACH\_ID"**

**entity-name="Coach"** cascade="persist,merge" />

**<bag name**="players" inverse="true" fetch="select"

cascade="persist,merge">

<key column="TEAM\_ID"/>

<one-to-many entity-name="Player" />

</bag>

<property name="nbLost" column="NB\_LOST" type="int"/>

<property name="name" column="TEAM\_NAME" type="string"/>

</class>

</hibernate-mapping>

Vous devez désormais utiliser une Map par entité :  
Map<String, Object> coach = new HashMap<String, Object>();

coach.put("name", "dynaCoach");

La subtililté est d’exploiter une session fondée le mode map : mapSession = s.getSession(EntityMode.MAP).

Pour récuperer des instances :

tm.begin();

Session s = (Session)em.getDelegate();

**Session mapSession = s.getSession(EntityMode.MAP);**

List results = mapSession.createQuery("from Team t ").list();

**Map team = (Map)results.get(0);**

tm.commit();

###### Mapping XML / Relationnel

La mapping XML/Relationnel remplit un fichier XML depuis la base de données en passant par la session Hibernate et vice versa.

Exemple avec la classe TEAM:  
<hibernate-mapping >

<class entity-name="Team" table="TEAM" **node**="team">

<id name="id" column="TEAM\_ID" type="long" node="**@id**">

<generator class="native"/>

</id>

<many-to-one name="coach" column="COACH\_ID"

entity-name="Coach" cascade="persist,merge" />

<bag name="players" inverse="true" fetch="select"

cascade="persist,merge" **embed-xml="true**">

<key column="TEAM\_ID"/>

<one-to-many entity-name="Player" />

</bag>

<property name="nbLost" column="NB\_LOST" type="int"

node="@nbLost"/>

<property name="name" column="TEAM\_NAME" type="string"

node="@name"/>

</class>

</hibernate-mapping>

Pour récuperer un document XML depuis une session, il faut récuperer la session en mode DOM4J.  
Session dom4jSession = session.getSession(EntityMode.**DOM4J**);

List elements = dom4jSession.createQuery("from Team").list();

for (int i=0; i<elements.size(); i++) {

**print( (Element) elements.get(i) );**

}

Le résultat obtenu est :  
<team id="3" name="Racing Club de Lens" nbWon="0" nbLost="0" nbPlayed="0">

<coach id="3" name="Fernandez"/>

<players>

<player id="3" name="utaka" number="0" height="0.0" weight="0.0"

hasBeenMvpCount="2">

<team id="3"/>

</player>

</players>

</team>

**embed-xml="true"** pour l’association ManyToOne depuis team vers coach

provoque l’insertion d’un élément XML détaillé dans l’arbre.

embed-xml="false" pour l’association ManyToOne depuis player vers team

provoque l’insertion d’un élément XML ne contenant que l’indication de jointure entre

les éléments (id).

La fonctionnalité XML -> relationnel est très interessante pour integrer des applications clientes alimentant la base de données ou pour exporter ses données au format XML.

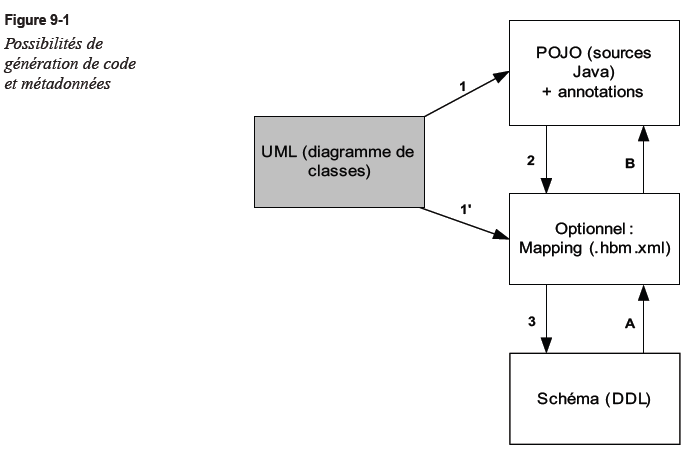
##### La suite Hibernate Tools

* comprend un éditeur de mapping pour faciliter l’écriture de fichier de mapping
* fonction de prototypage des requetes
* Schema export permet de générer rapidement un schéma de base de données

2 approches sont possibles :

Modèle relationnel existant -> départ de la conception fonctionnel

Enrichissement d’applications existantes -> départ de la base de données



Depuis la base de données ( DDL) :

Générateur de XML **Hibernate XML Mappings** pour la génération des fichiers hbm.xml.

Génération de code **Domain code** pour la génération des classes persistante et des annotations.

Depuis UML :

Génération de fichiers de mapping via l’atelier UML ou **AndroMDA**

Génération du schéma de base de données: **SchemaExport**

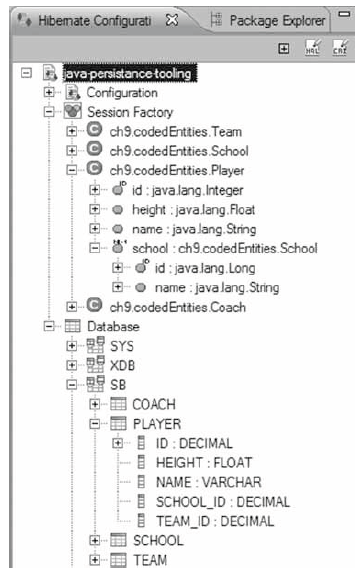
Hibernate tools peut s’utiliser comme tache ant en important le **jar** hibernate-tools.jar  
On peut aussi l’integrer à un IDE comme Eclipse sous forme de **plugins**

Red Hat Developper Studio intègre Hibernate, SEAM

###### Fonctionnalités:

L’éditeur de fichiers de mapping colore les champs, propose l’autocompletion et corrige les noeuds incomplets.

**Hibernate Console**: pour être utilisé, il faut d’abord créer une configuration hibernate console (en selectionnant le projet -> New -> Other -> Hibernate).  
On obtient un ensemble de vue s



Lorsque on déplie la racine, une sessionFactory dédié à la vue est crée .

Configuration permet d’explorer les entités par leur nom.

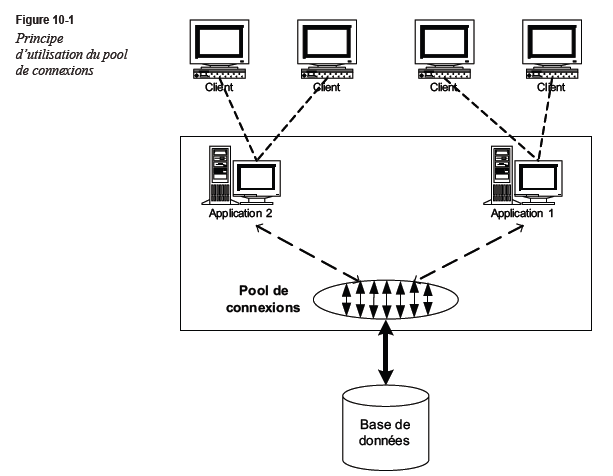
SessionFactory autorise la navigation par nom de classe qualifié.

Database permet de visualiser la structure de la base de données.

Page 325

##### Projets Annexes JPA/Hibernate

L’établissement d’une connexion JDBC est couteuse en performance. Le pool de connexion  reste ouvert vers la base de données en permanence, et est utilisé par les clients. Le pool permet de controler le nombre de connexions ouverte, la durée de vie , la durée de veille maximale.



###### Gestion des pools de connexion

C3Po :fiable et propose des possibilités de paramétrages étendues. A définir dans le fichier persistance.xml, et utiliser un fichier séparé c3po.properties, à ajouter au CLASSPATH.  
Nous pouvons configurer à l’intérieur du fichier c3po.properties la taille min et max du pool, la durée d’attente maximal pour l’acquisition d’une connection, le délai entre chaque essai, l’autocommit lors de la fermeture , etc…

Proxool : proposé par défaut par Hibernate. Même paramètre mais dispose d’une servlet permettant de visualiser l’état du pool de connexion : <http://proxool.sourceforge.net>.

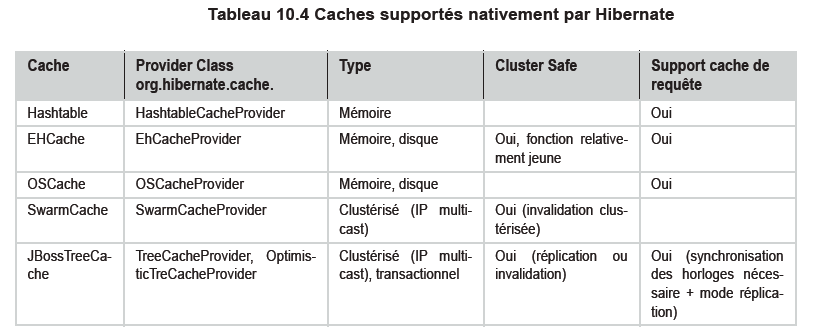
###### Utilisation du cache de second niveau :

Le cache de premier niveau est le gestionnaire d’entité, qui a une durée de vie courte et n’est pas partageable entre plusieurs traitements. Plusieurs types de cache second niveau sont disponibles :

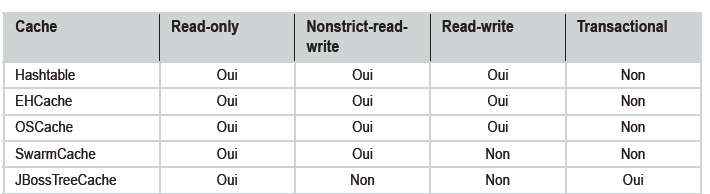
Moins le cache est utilisé en écriture, plus il est efficace, notamment sur les données les plus consultés ( de paramétrage tel que les pays, les codes postaux… ). Ne pas utiliser si on utilise un cluster ou une répartition de charge sur plusieurs machines (rafraichissement régulier et temps de latence).

On trouve des **caches d’entités**, ou des **caches de collections**, relié à des recherche par id.

Il existe plusieurs implémentations des caches, supportant les requêtes, cluster et mémoire disque :



Les stratégies de caches sont:  
Lecture seule : cas ou l’appli ne fait que lire les données, sure même dans un cluster.  
Lecture – Ecriture : cas ou l’appli lit et écrit. Ne pas utiliser si cluster ou si l’appli nécessite un niveau d’isolation transactionnelle serialisable.  
Lecture – Ecriture non stricte : maj occassionnelle, avec peu de risque qu’il y ait collision transactionnelle.  
Transactionnelle : les transactions sont assurés, mais uniquement dans un environnement JPA.



Pour mettre en place un cache il faut ajouter son archive au classpath, determiner les entités à mettre en cache, et mettre à jour le persistance.xml :

<property name="hibernate.cache.provider\_class"

value="org.hibernate.cache.EhCacheProvider"/>

EhCache dispose d’un fichier de configuration spécifique à mettre dans « resources », permettant de configurer les classes à être mis en cache, la durée de veille, le nombre maximal d’instances, la durée de vie maximale.  
On pourra vérifier le démarrage du cache de second niveau dans les traces de démarrage d’Hibernate :  
INFO SettingsFactory:259 - Cache provider:

org.hibernate.cache.EhCacheProvider

INFO SettingsFactory:186 - Second-level cache: enabled

Dans vos classes java, utilizer l’annotation **@org.hibernate.annotations.Cache** , non spécifique à un cache en particulier , prenant 3 membres :  
usage : mise en place de la stratégie  
region (opt): région de cache,   
include (opt): all ou non-lazy pour exclure les proprietés lazy.

La modification d’une entité mis en cache par la statégie read-only génère une exception. Si on utilise la statégie non strict read write, il n’y aura plus d’exception, en revanche la maj de l’entité aura pour effet de la supprimer du cache de second niveau, ce qui entrainera un select pour la récuperer à nouveau ( aucun interet donc de mettre en place un cache de second niveau pour les entités frequemment modifiés).

Contrôle de l’interaction avec le cache :

En passant par la session hibernate, on peut configurer les interactions avec le cache :  
session.setCacheMode(CacheMode.IGNORE);

Les attributs sont :  
**NORMAL** : récupère les objets du cache niv 1 (gestionnaire d’entité), et les place dans le cache niv 2.

**GET** : récupère les objets du cache niv 2 mais n’y accède pas en écriture, sauf lors de la maj de la BD.  
**PUT** : récupère les objets depuis la BD et non depuis le niv2, mais écrit dans le niv2.  
**IGNORE** : ignore le cache hormis pour effectuer l’invalidation lors des mises à jour.  
**REFRESH** : idem à GET, tout en ignorant l’effet du paramètre hibernate.use\_minimal\_puts ce qui force le rafraichissement du cache de second niveau pour toutes les données lues en base de données.

###### Le cache de requête

Les classes , les collections , et les requetes sont trois notions différentes du point de vue du cache. Les requetes demandent des traitement spécifiques pour bénéficier du cache et pour être utilisé conjointement avec les classes et les collections.

Exemple :

@Entity

**@org.hibernate.annotations.Cache(usage =**

**org.hibernate.annotations.CacheConcurrencyStrategy.TRANSACTIONAL)**

@org.hibernate.annotations.NamedQuery(name="myNamedQuery",

query="Select team from Team team where team.name = :param ",

cacheable=true

)

public class Team {…}   
Il faut activer le cache des requetes dans persistance.xml :

<property name="hibernate.cache.use\_query\_cache" value="true"/>

Le cache de requete n’a de sens que pour une même requete, avec les mêmes valeurs pour les paramètres injectés, et en association avec le cache d’entités.

###### Réplication de cache avec JBoss

Dans le cadre de cluster, chaque machine disposera d’un cache, la modification d’une entité doit se propager dans tous les caches.