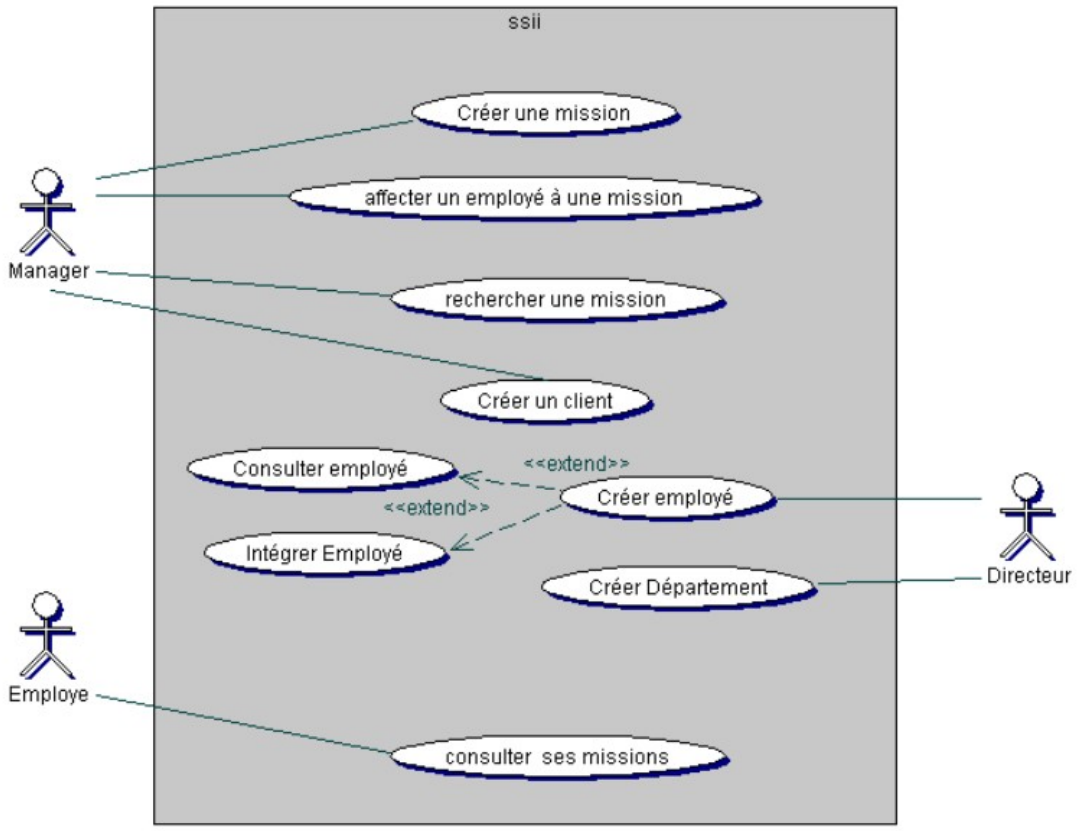
Présentation des TPs

Le but de cette série de TPs est de créer une application mettant en œuvre le *framework* Hibernate de manière progressive. On partira d'un exercice de rappel sur JDBC pour peu à peu mettre en œuvre la majeure partie des fonctionnalités d'Hibernate exposées dans le cours.
On commence par présenter le sujet du TP : une gestion simplifiée des collaborateurs d'une SSII.

Présentation de l'application

Le but de notre système logiciel SSII est de gérer de manière simplifiée les missions et les employés affectés à celles-ci.

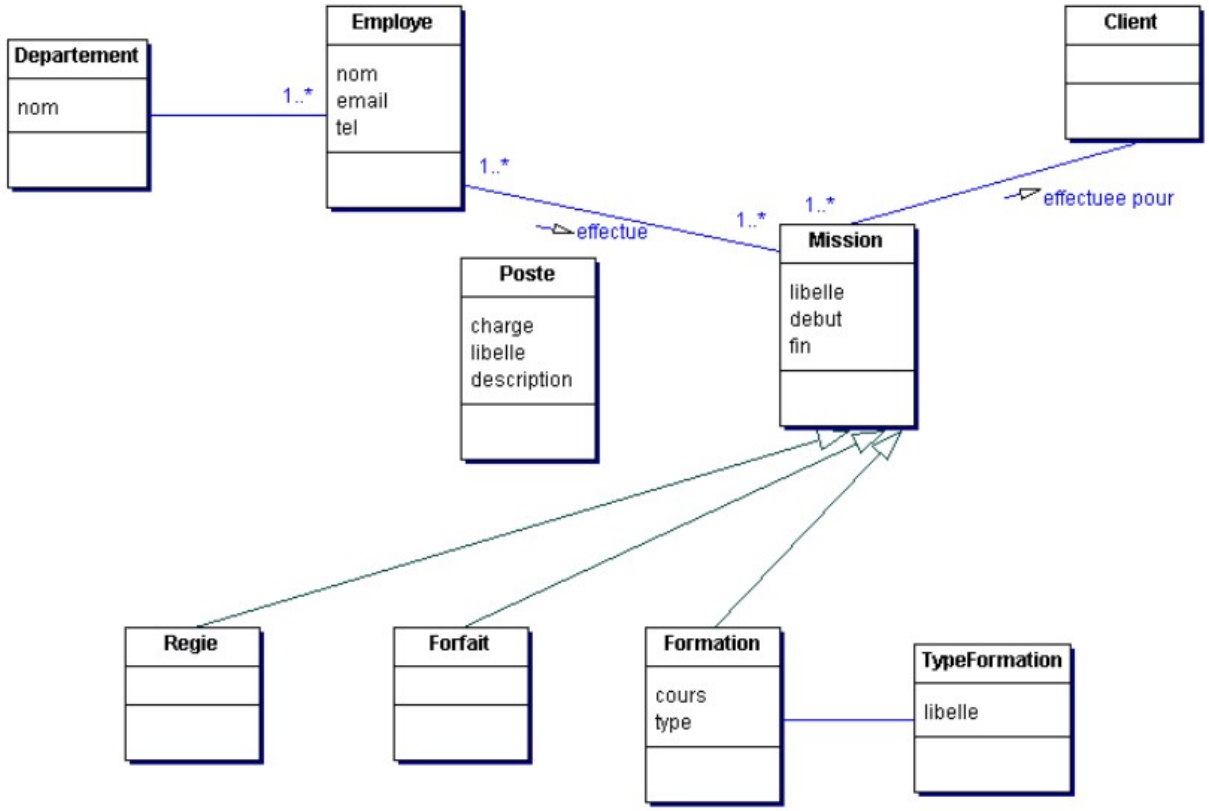
Le diagramme de cas d'utilisation suivant illustre le besoin couvert :



Trois acteurs sont identifiés :

- Le directeur en charge de l’embauche et donc de la création des employés ainsi que de l’organisation de sa société.
- L’employé qui souhaite pouvoir consulter les missions qu’il effectue et a effectuées.
- Le manager qui doit pouvoir créer des missions et affecter celles-ci aux employés.

L’analyse fait apparaître le modèle du domaine suivant :



La société est organisée en départements comportant des employés. Ceux-ci peuvent effectuer plusieurs missions et le travail dans une mission est défini par un poste. Sur une période de temps, un employé peut avoir plusieurs missions qu’il effectue tant que sa charge de travail ne dépasse pas 5j / semaine. Le système devra vérifier cette règle lors de l’affectation d’un employé. Il y a plusieurs types de mission :

- La régie qui se déroule généralement chez le client et peut être une régie à temps partiel.
- Le forfait qui est un projet réalisé en interne par la société.
- La formation qui peut être soit une écriture, soit une animation de cours.

TP1 SSII : Rappel JDBC

Ce TP va se focaliser sur les classes *Employe* et *Departement*. On va écrire le code nécessaire à la réalisation des cas d'utilisation suivants :

- Consulter la liste des employés d'un département.
- Créer un employé.
- Intégrer un employé dans un département.

Le détail des cas d'utilisation est le suivant :

Consulter la liste des employés d'un département
1 : L'utilisateur fournit le nom du département.
2 : Le système recherche le département portant ce nom est les employés travaillant pour lui, puis les affiche.

Créer un employé
1 : L'utilisateur fournit le nom, l'email et le téléphone de l'employé.
2 : Le système crée un nouvel employé et l'affiche en donnant son ID.

Intégrer un employé dans un département
1 : Le système affiche la liste des départements et leur ID.
2 : L'utilisateur fournit l'ID de l'employé et l'ID du département d'affectation.
3 : Le système intègre l'employé dans le département et affiche la nouvelle liste des employés du département.

Afin d'organiser proprement notre code nous suivrons le pattern DAO avec les classes suivantes :

- ***Departement*** et ***Employe*** sont les classes métier.

- ***DBHelper*** est une classe utilitaire pour gérer la connexion à la base de données.
- ***DptDAO*** est la classe gérant l'accès aux données pour les départements et les employés.
- La classe ***TP1Test*** est un TestCase (framework Junit) qui implémente les trois cas d'utilisation que nous aurons à tester.

Le travail demandé consiste à terminer l'écriture des classes dont le squelette est fourni. Commencez par le premier CU (voir TP1Test) puis le second, etc. Les tests ont été construits pour se dérouler dans l'ordre donné compte tenu d'une base correctement initialisée. Utiliser le client de base de données pour jouer le script de création et d'initialisation de la base : TP<*>.sql.
Rejouer ce script entre chaque exécution du test pour être sûr de repartir d'une base initiale.

1) Complétez la classe *DBHelper* avec le nom de la base précédemment créée et initialisée
2) Complétez la classe *DptDAO* en réalisant les services dans l'ordre indiqué qui reprend celui des tests. Pour tester de suite mettez en commentaire les CU qui ne sont pas encore complétés.

TP2 SSII : Démarrez en Hibernate

Ce second TP reprend ce qui a été réalisé avec le premier TP mais remplace l'usage de JDBC par celui d'Hibernate. L'organisation du code est la même et on retrouve les classes suivantes :

- ***Departement*** et ***Employe*** sont les classes métier.
- ***DBHelper*** est une classe utilitaire pour gérer la connexion au moteur Hibernate (*SessionFactory*).
- ***DptDAO*** est la classe gérant l'accès aux données pour les départements et les employés. Cette classe s'appuie sur Hibernate.
- La classe ***TP2Test*** est un TestCase (framework Junit) qui implémente les trois cas d'utilisation.

1) Complétez le fichier de mapping ***hibernate.cfg.xml*** et la classe ***DBHelper*** pour obtenir une SessionFactory.
2) Complétez les classes métier ET leur fichier de mapping associé.
3) Complétez la classe ***DptDAO*** en utilisant les requêtes HQL fournies. Faites-le dans l'ordre donné en mettant en commentaire les CU que vous n'aurez pas encore réalisés ceci afin de tester dès qu'un CU est terminé.

TP3 SSII : Démarrez avec JPA

Ce 3ème TP reprend ce qui a été réalisé avec le 2 premiers TP mais utilise JPA. L'organisation du code est la même et on retrouve les classes suivantes :

- *Departement* et *Employe* sont les classes métier.
- *DBHelper* est une classe utilitaire pour initialiser JPA et fournir un *EntityManagerFactory*.
- *DptDAO* est la classe gérant l'accès aux données pour les départements et les employés. Cette classe s'appuie sur JPA.
- La classe *TP3Test* est un TestCase (framework Junit) qui implémente les trois cas d'utilisation.

1. Complétez le fichier de ***persistence.xml*** et la classe ***DBHelper***.
2. Complétez les classes métier ET utiliser les annotations JPA.
3. Complétez la classe ***DptDAO*** en utilisant les requêtes JPQL fournies. Faites-le dans l'ordre donné en mettant en commentaire les CU que vous n'aurez pas encore réalisés ceci afin de tester dès qu'un CU est terminé.

Dans un second temps, modifier la propriété ***hbm2ddl.auto*** en mode *create* et mettre au point un fichier *import.sql* pour initialiser la base avec des données de test

TP4 : Mapping de l'héritage

Ce troisième TP se focalise sur l'arbre d'héritage des missions de la SSII que l'on rappelle ci-dessous.

Le TP se déroule en deux étapes :

- Dans la première étape, il s'agit de compléter le code fourni pour réaliser le mapping en utilisant la stratégie « une table par classe ».
  On a donc en base les tables ***tmission***, ***tformation*** et ***tregie***.
  On ne traite pas la classe Forfait pour le moment.
- Dans la seconde étape, il faut prendre en compte la classe *Forfait* mais aucun code n'est fourni.

La démarche pour l'étape 1 est la suivante :

- Complétez les classes ***Mission***, ***Formation*** et ***Regie***. (Attributs et annotations)
- Complétez la classe *MissionDAO* pour les cas suivants :
  - Lister toutes les missions.
  - Lister toutes les régies.
  - Créer une nouvelle régie.

Ensuite, enrichissez l'exercice de la classe Forfait.

On ne prend en compte que les classes de la hiérarchie, les associations seront traitées ultérieurement.

TP5 SSII : Les associations

Ce TP comporte trois étapes :

- Mapping d'une relation *one-to-many* entre client et mission en utilisant une liste.
- Mapping de la relation *many-to-one* entre une mission et un client.
- Mapping de la classe-association Poste présente entre *Employe* et *Mission*.

Le TP5 se base sur le TP4 (mapping de l'héritage).

Étape 1 : Relation one-to-many Client / Mission avec utilisation d'une list

Dans cette étape :

- Complétez la classe Client pour utiliser une liste de missions.
- Annotez la pour définir une relation uni-directionelle
- Effectuez la méthode de test *TP5_1Tests/testConsulterClients()*.

Combien de requêtes sont effectuées ? A quel moment ?

Étape 2 : Relation inverse many-to-one Mission / Client

On décide de pouvoir aussi naviguer d'une mission vers son client.

- Complétez la classe ***Mission***.
- Annoter en conséquence pour définir une relation bi-directionelle
- Implémenter et exécuter toutes les méthodes de tests de *TP5_1Tests*.

Étape 3 : Relation *ManyToMany*

Mettre en place une relation *ManyToMany* bi-directionelle entre ***Employe*** et ***Mission***. Utilisant la table ***tposte*** comme table d'association

Étape 4 : Classe association Poste

On a rajouté la table ***tposte*** pour modéliser la classe-association ***Poste*** entre ***Employe*** et ***Mission***. On veut que les navigations suivantes soient permises :

- Un employé connaît ses missions.
- Un poste connaît son employé et sa mission.
- Une mission connaît ses employés.

Dans cette étape :

- Complétez la classe *Poste*.
- Complétez les classes *Employe* et *Mission*.
- Annoter en conséquence.
- Implémenter et exécuter les tests de TP5_2Tests après avoir réinitialisé la base

TP6 SSII : la persistance transitive

Ce TP va se dérouler en deux étapes :

1. La mise en place d'un objet valeur pour l'adresse email d'un employé.
2. La mise en place d'une persistance transitive entre la classe *Forfait* et la nouvelle classe *Tache*.

Étape 1 : Objet Valeur

En effet, une adresse email doit savoir renvoyer son nom de domaine, etc. En objet, l'adresse email est donc plus qu'une simple String. On en fait une classe Email dont chaque Employe possède sa propre instance. En base l'adresse email reste sauvegardée dans la table Temploye.

- Complétez les classes *Employe* et Email.
- Annoter en conséquence.
- Exécutez le test : TP6_1Tests.
- Complétez dans celui-ci le test du CU : créer et modifier l'adresse d'un employé.

Étape 2 : Cascading

On va rajouter à notre modèle la classe Tache qui représente les unités de travail qui découpent un projet au forfait. Une nouvelle table ***ttache*** contient les informations d'une tâche (libellé et charge). Seule la classe Forfait se voit mettre en place une relation de composition 1-N vers Tache (le cycle de vie d'une tâche est lié à son forfait). Cette association sera représentée par une Map dont la clé est le nom de la tâche et la valeur, la tâche elle-même.
Plusieurs tests sont mis en place pour :

1. Afficher les forfaits et tâches,
2. Modifier une tâche,
3. Créer une tâche et la rendre persistante par propagation depuis le forfait,

4. Retirer une tâche d’un forfait et vérifier qu’elle est bien détruite,
5. Créer un forfait et ses tâches en ne rendant persistant que le forfait qui propage cette propriété à ses tâches,
6. Supprimer un forfait ce qui entraîne la suppression de ses tâches.

Pour vérifier que les tests passent, utilisez le client SQL pour inspecter les tables.

Dans cette étape :

- Complétez la classe Forfait pour ajouter la Map et les méthodes de gestion de tâches.
- Annoter *Forfait.java* pour déclarer l’association en utilisant une Map.
- Complétez les tests de la classe TP6_2Tests.

TP7 : Accès aux données

Ce TP demande d'écrire les requêtes HQL permettant de récupérer les données demandées. Il se déroule en 3 étapes.

Étape 1 : Lazy-loading

Le chargement d'un département ne provoque pas le chargement de ses employés (lazy= « true ») par défaut.
Écrire les méthodes de tests suivantes :

- Un test qui déclenche une *LazyInitializationException*
- Préchargement par JPQL
- Utilisation de *Hibernate.initialize*
- Utilisation de profils de fetch Hibernate
- Définition d'un *EntityGraph*

Étape 2 : JPQL/HQL

Écrire les requêtes JPQL/HQL (TP7_2Tests) permettant d'effectuer les recherches suivantes :

- Retrouver en une seule requête l'ensemble des noms des employés et des départements.
- Retrouver les employés affectés à la mission de libellé « Formation CASA ».
- Retrouver les employés travaillant pour le client « *Computing corp* ».
- Afficher tous les postes ayant une charge inférieure à 3j/semaine ainsi que l'employé et la mission concernés.
- Retrouver tous les employés qui ne sont pas affectés.
- Supprimer tous les postes de la mission « Formation CASA »

Étape 3 : Criteria API

- Afficher tous les postes ayant une charge inférieure à 3j/semaine ainsi que l’employé et la mission concernés.
- Charger les départements et les employés associés via *l’EntityGraph* précédemment défini

TP8 : Transactions

Ce TP a pour objectif d'illustrer les concepts relatifs au contrôle de concurrence des transactions.

Gestion optimiste

La classe TP8_1Tests propose deux cas d'utilisation permettant d'insérer un nouvel employé dans la base et de simuler 2 transactions concurrentes essayant de mettre à jour le même enregistrement.

1. Mettre en place une gestion optimiste de la classe *Employe*.
2. Exécuter le cas de test de création et observer le résultat dans la table.
3. Écrire le cas de test permettant de simuler 2 transactions concurrentes.
4. Exécuter le cas de test
5. Implémenter une stratégie optimiste pour traiter le problème de concurrence

Transaction utilisateurs

Cette partie a pour objectif de réaliser une transaction utilisateur, s'étalant sur plusieurs transactions physiques :
Affecter un employé à une mission.

La classe TP8_2Test propose le test de deux scenarii :

- Le cas où l'employé sélectionné sera surbooké si on lui affecte cette mission (il suffit de sélectionner Jean Dupond).
- Le cas où tout se passe bien (Henri Prévost). Pour les sélectionner ils sont fournis dans l'ordre de la base dans la liste résultat.

Chaque scénario comporte trois requêtes avec interaction avec l'utilisateur :

- L’affichage de la liste des missions pour en sélectionner une.
- L’affichage des employés qui n’ont pas déjà un poste sur cette mission.
- L’affectation de l’employé à la mission avec vérification d’un surbooking éventuel. Cette vérification s’effectue dans la classe CreerPosteRM qu’il vous faut compléter.

Complétez la classe TP7Test et mettez en place les classes DAO nécessaires.

TP9 : Optimisations

Monitoring
Activer le monitoring
Exécuter le test TP9_1Test et visualiser la console

Afficher via l'API, l'objet *Statistics*

Batch
Exécutez la méthode *testOutOfMemory* avec un Java Heap de 64M (Option ***-Xmx64M***), vous devez observer un *OutOfMemoryException*
Créez une autre méthode *testBatch*, configurez Hibernate optimiser le code précédent et éviter *l'OutOfMemory*.

Cache de second niveau
Configurer l'implémentation *EHCache*
Définir un cache pour la hiérarchie *Mission*
Écrire une nouvelle classe de test qui démontre :

- L'utilisation du cache :
- Que les collections ne sont pas mises en cache par défaut
  - Ensuite, mettez à jour la configuration et mettez en cache les collections,
- Qu'une insertion, n'écrit pas dans le cache
- Qu'une mise à jour écrit dans le cache
- Le comportement de *CacheMode.GET*

Utiliser l'API de statistiques pour montrer l'usage du cache et autres informations

EntityListener

Créer une classe de Logging qui implémente *SaveOrUpdateEventListener* et *LoadEventListener*

Enregistrer la classe Logging comme listener d’évènements Hibernate