Analyse et conception objet par la pratique

À propos des auteurs:

Stéphane ALONSO ingénieur de formation, professeur, enseigne UML et les technologies objets.

Dino WAZIR ingénieur de formation, professeur agrégé, enseigne l'informatique industrielle et les technologies objets en particulier sur la plate forme Java.

Première partie – Analyse UML

Introduction

Le but de cette série d'articles est d'appréhender à travers la réalisation d'une application web de gestion d'une médiathèque, un ensemble de techniques employées depuis l'analyse du cahier des charges en passant par la conception, et jusqu'au codage et la validation finale d'une l'application.

La programmation sera fortement orientée objet avec l'utilisation du framework Java et plus particulièrement de la dernière version pour la réalisation d'application Web : JSF (Java Server Face).

Ce premier article va nous permettre de planter le décor: présentation du projet, du cycle de développement, du langage UML et enfin analyse de notre cahier des charges.

L'IDE utilisé, (disponible pour Linux et Windows) est NetBeans 6.0 (http://www.netbeans.org) avec le module de développement Web.

Pour le modélisation UML, de nombreux outils libres existent, soit sous forme autonome comme ArgoUml, Bouml, Papyrus, StarUML, Umbrello ou intégré à des IDE avec Netbeans et Eclipse.

Médiathèque : Cahier des charges

Voici énoncé de manière sommaire les caractéristiques et autres fonctionnalités de notre future médiathèque:

1. La médiathèque est gérée par un ou plusieurs gestionnaires;
2. La médiathèque est accessible à un certain nombre d'adhérents;
3. Un adhérant doit s'acquitter d'une cotisation pour une certaine durée d'utilisation;
4. La médiathèque possède un catalogue mis à jour par les gestionnaires;
5. Le catalogue décrit l'ensemble des médias présents dans la médiathèque;
6. On peut effectuer des recherches multi-critères dans le catalogue de la médiathèque;
7. Les médias sont classés en plusieurs catégories: média papier, sonore et vidéo;
8. Un adhérant peut emprunter de 1 à 3 médias maximum;
9. Un média doit être rendu au maximum au bout d'une durée défini, par exemple: un mois pour les médias papier et sonore, et une semaine pour les médias vidéo;
10. Seuls les gestionnaires peuvent traiter les prêts de média;
11. Les adhérents disposent d'un certain nombre de postes pour consulter le catalogue et sélectionner les médias à emprunter;
12. Un gestionnaire peut, le cas échéant, être lui-même un adhérent de la médiathèque. C'est le

cas, par exemple, d'une médiathèque gérée par une association dont les gestionnaires sont eux-mêmes des membres désignés.

Notre médiathèque sera une application de type Web. L'architecture matérielle comprend un ou plusieurs postes de consultation pour les adhérents, un poste de gestion des entrées/sorties et un serveur (serveur web et serveur de données).

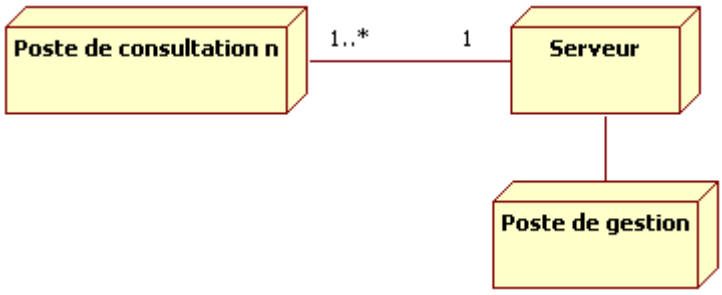
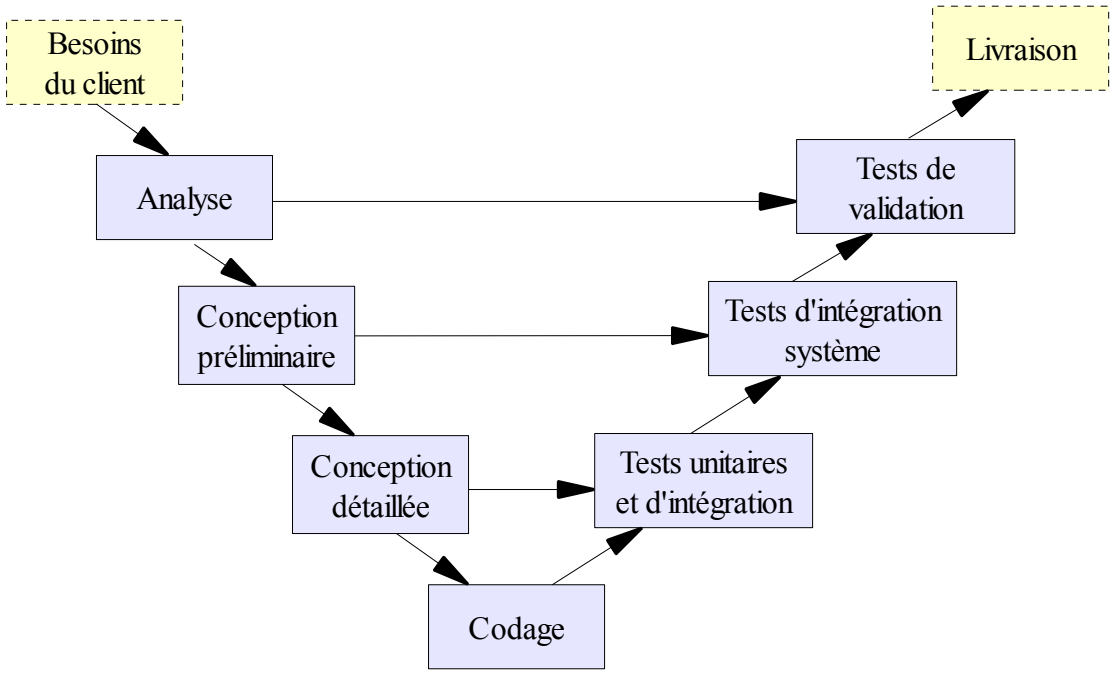


Diagramme de déploiement UML

Cycle de développement d'un logiciel

Le développement de logiciels professionnels est réalisé en suivant un cycle de développement appelé cycle en V à cause de sa forme. Dans sa partie descendante, il décrit l'ensemble des étapes à réaliser avant codage. Dans sa partie ascendante se trouve l'ensemble des tests de vérification de chacune des étapes précédentes.



Exemple de cycle en V

Description des phases

ANALYSE

Partant du cahier des charges (parfois incomplet et très imprécis), il est nécessaire de bien déterminer les besoins du client.

La phase d'analyse permet de s'accorder sur ce que doit faire le système, le « quoi faire », avant de de passer à la phase de conception pour déterminer la manière dont il doit le faire, le « comment faire ».

L'analyse fonctionnelle, c'est la recherche:

- des ***fonctionnalités*** offertes par le système;
- des ***éléments qui constituent*** le système;
- mais aussi des ***éléments extérieurs*** (opérateur humain, matériel) avec lesquels le système va ***communiquer***;
- des ***échanges*** entrent l'ensemble de ces éléments;
- de la description sous forme de ***scénario*** de l'utilisation du système.

Conception préliminaire

Partant de l'analyse définitive, on réalise une ébauche de la structure du système (***la manière dont il doit le faire, le « comment faire »***), mais sans contrainte technologique forte, on doit ici faire le plus possible abstraction des spécificités des composants matériels et logiciels nécessaires (par exemple la base de donnée qui sera utilisée ...).

Une fois cette ébauche réalisée, des choix technologiques devront être faits.

Mise en oeuvre / prototypage

Une phase de mise en oeuvre peut être nécessaire pour permettre à l'équipe de développement d'acquérir la maîtrise des solutions technologiques retenues avant d'aborder la conception détaillée de l'application, elle permet également de détecter suffisamment tôt d'éventuels problèmes liées aux choix retenus.

Conception Détaillée

La conception détaillée est le « *comment c'est fait* ». Cette phase peut être dépendante des choix technologiques (langage utilisée ...). Ne perdons pas de vu que le code source qui sera produit en phase de codage devra représenter fidèlement la modélisation UML réalisée pendant cette phase de conception détaillée, contrairement à la phase d'analyse où le modèle UML est un méta-modèle représentant ce que doit être l'application du point de vue des utilisateurs.

De façon à prévoir toute évolution de l'application suite à une modification du cahier des charges, on pourra adopter un développement modulaire en s'appuyant sur la méthode dite n-tier (3-tier dans sa forme classique). L'approche objet mettra en oeuvre les motifs de conception (design pattern) appropriés (Factory, Observer, Singleton, etc..).
Nous verrons dans la partie conception de notre médiathèque comment utiliser cette méthode en détail.

Codage / Réalisation

Dans cette passe il ne reste qu'à écrire les algorithmes des méthodes de chaque classe décrite dans la phase précédente.

La phase de codage permet de s'accorder sur le « *comment c'est fait* ».

L'ensemble des algorithmes est codé dans le langage (ou les langages) retenu pour le développement.

On pourra remarquer que le codage vient tardivement dans le processus de développement. En effet, comme dit fort bien La Fontaine « Rien ne sert de courir, il faut partir à point ». Une bonne analyse est synonyme de chance de succès alors que se précipiter sur le clavier ne garantit que l'échec, surtout aujourd'hui avec des applications de plus en plus complexes et avec des équipes de

développement de plus en plus grandes.

Les tests

L'ensemble du logiciel est testé tout le long du cycle de développement.

À chaque phase correspond un ensemble de tests spécifiques, de la simple vérification de la validité d'un algorithme à la dernière phase avant la distribution du logiciel, les tests de validation qui permettent de savoir si le produit correspond bien aux attentes du client.

Et UML dans tout cela

Afin de réaliser les phases d'analyse et de conception, nous allons utiliser le langage UML.

UML pour ***Unified Modeling Language***, est un langage graphique de modélisation adapté aux technologies objets.

Nous allons voir comment « lire » un diagramme UML, pré-requis essentiel avant de s'attaquer à l'analyse de notre système.

Décrypter les diagrammes UML

UML est un langage graphique de modélisation.

Cela signifie, qu'un élément graphique peut être remplacé par une phrase, le tout étant de savoir quelles sont les significations de ces éléments graphiques.

Il y a une hiérarchie des niveaux de signification des éléments graphiques dans UML, comme:

- ✔ Le type de diagramme
- ✔ Les éléments représentant des objets dans le diagramme
- ✔ Les éléments de liaison entre les objets

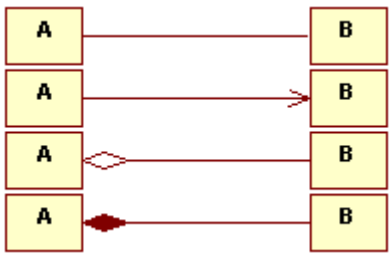
Éléments de liaison inter-objets

Ils permettent comme leur nom l'indique de visualiser le lien entre deux entités présentes sur un même diagramme.

Pour cela on distingue trois familles de liens possibles, chaque famille pouvant avoir plusieurs déclinaisons plus spécifiques.

En effet deux entités A et B peuvent avoir les liens suivants:

- ✔ A ***« est associé à »*** B : représenté par un lien en trait plein avec ajout d'attributs graphiques optionnels (flèche simple, losange plein ou vide à une extrémité) ;

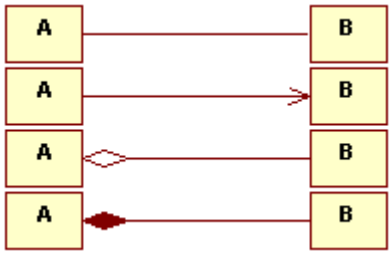


La famille des associations

- ✔ A ***« est une sorte de »*** B : représenté par un lien en trait plein ou en pointillé avec à une extrémité une flèche en forme de triangle ;

La famille des héritages

- A ***« est associé à »*** B : représenté par un lien en trait plein avec ajout d'attributs graphiques optionnels (flèche simple, losange plein ou vide à une extrémité) ;



La famille des associations

- A ***« dépend de »*** B : représenté par un lien en trait en pointillé avec une flèche simple à une extrémité.

La dépendance

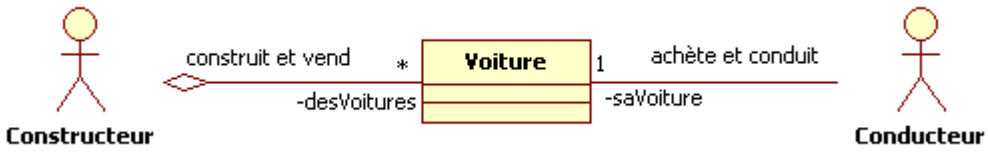
Ces liens peuvent aussi être stéréotypés. Un stéréotype est une information textuelle ajoutée sous la forme suivante : << stéréotype >>. Ils permettent d'ajouter une information qui ne possède pas de symbole graphique particulier, ou pour faciliter la lecture (ceci afin de ne pas multiplier indéfiniment le nombre de symboles graphiques).

Exemples :

Le constructeur automobile

Un constructeur automobile construit et vend des voitures. Un conducteur achète sa voiture au constructeur et il la conduit. Il peut la revendre, ou la prêter. Si le constructeur fait faillite (il n'existe plus) les voitures déjà construites continuent d'exister.

En UML cela se traduit par le diagramme ci-dessous:



Les éléments **Constructeur** et **Conducteur** sont appelés des **acteurs**, ce ne sont pas des éléments uniques, ils représentent l'ensemble des éléments ayant les mêmes propriétés (tous les constructeurs automobiles et tous les conducteurs de voiture).

L'élément **Voiture** est appelée une **classe**, elle représente l'ensemble des voitures, qui elles sont des **instances de classe** ou **objets**.

Les liens entre les éléments **Constructeur** et **Voiture** d'une part et entre **Conducteur** et **Voiture** d'autre part son appelés des **associations**. On peut remarquer une différence entre les deux associations présentes sur ce diagramme.

- Il y a un losange blanc sur l'association côté constructeur. Cela signifie qu'il est responsable de la naissance de la voiture, mais qu'il peut la céder ou la prêter à une autre entité du diagramme. De plus, sa disparition est sans incidence sur les voitures déjà construites. Dans ce cas on parle **d'agrégation par référence**.
- L'étoile signifie qu'un constructeur construit un nombre indéfini de voitures. On parle de multiplicité, lorsqu'elle n'est pas indiquée cela veut dire 1.
- Le conducteur utilise la voiture qu'il n'a pas construite, il peut lui aussi la céder à quelqu'un d'autre avant sa mise à la casse (sa destruction). On parle ici d'association simple.

Les tours d'immeubles

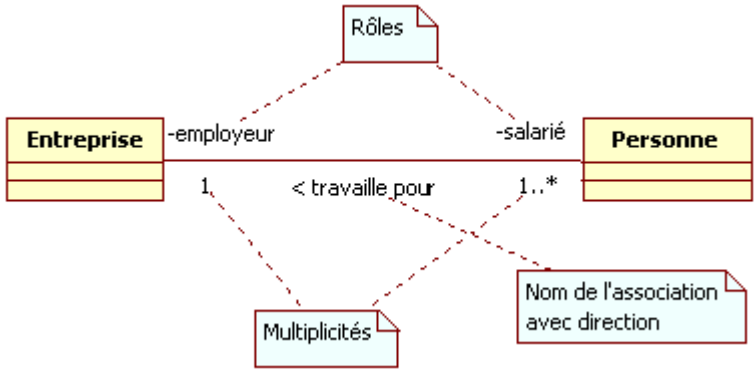
On construit un ensemble d'immeubles constitués chacun de plusieurs appartements. On ne peut évidemment pas transférer un appartement d'un immeuble à l'autre. De plus, si on détruit l'immeuble, l'ensemble des appartements de cet immeuble sont eux aussi détruits, on dit qu'il y a correspondance de vie ou propagation du destructeur.

En UML cela se traduit par ce diagramme :

L'apparition du losange noir transforme la simple association en **agrégation par valeur**.

Remarque : être responsable de la naissance d'un objet se traduit en langage informatique par l'emploi de l'opérateur **new** (Java et C++) ou l'appel implicite du constructeur sans argument (C++) au sein de la classe responsable.

Résumons les associations



Une association peut être précisée par différents éléments, mentionnons ici :

- ✓ L'association nommée avec une direction
- ✓ Les rôles
- ✓ Les multiplicités (ou cardinalités)

L'association peut toujours être nommée pour compléter la description de l'association. Dans l'exemple, ***une Personne «travaille pour» une Entreprise***. Notez que la direction de la lecture peut être affichée à l'aide d'une flèche à côté du nom de l'association.

Les rôles peuvent désigner la participation d'une classe dans l'association. ***Il est plus significatif de spécifier les rôles plutôt que le nom de l'association***.

Les cardinalités représentent une ***contrainte énumérative sur une association***. Dans la notation UML, les cardinalités se lisent comme suit :

- ✔ Une Personne travaille pour 0 ou 1 Entreprise;
- ✔ une Entreprise peut avoir de 1 à n Personnes employées.

Les éléments reliés via une association peuvent être n'importe quel élément graphique présent dans UML.

Nous développerons les autres concepts UML lorsque nous en aurons besoin tout au long de la réalisation de la médiathèque.

La médiathèque

Analyse UML

Cas d'utilisations

Le diagramme de « cas d'utilisation » est un des diagrammes UML qui permet de mettre en évidence les fonctionnalités interactives d'un système ainsi que les éléments extérieurs qui interagissent avec le système.

En résumé, les cas d'utilisations, décrivent le système du point de vue de l'utilisateur. Leur but est de ***capturer*** et ***identifier*** les besoins tout en décrivant une fonctionnalité du système en réponse à la stimulation d'un ***acteur*** externe.

Les acteurs

Personnes qui vont être amenées à utiliser l'application: clients, gestionnaires, personnels de maintenance, etc. .

Ou bien ***des systèmes externes qui vont interagir sur le logiciel***: Imprimante, Serveur, etc. .

Les acteurs doivent être identifiés et décrits, pour mettre en évidence leur rôle réel.

Un acteur peut aussi bien désigner une personne ou un système unique, ***qu'un groupe*** de personnes ou de systèmes, ayant ***des traits communs***.

Ici nous aurons l'ensemble des adhérents ainsi que les gestionnaires de la médiathèque. De plus, on suppose qu'un gestionnaire peut agir en tant d'adhérent pour se subsister à lui dans certaines situations. Ainsi un gestionnaire va ***« hériter »*** des caractéristiques d'un adhérent, on peut dire qu'un gestionnaire ***« est une sorte d' »*** adhérent. Les médias (livre, CD, DVD, etc..) gérés par la médiathèque sont eux aussi des acteurs.

Les cas d'utilisations

Voici comment on représente un cas d'utilisation.

Les cas d'utilisations servent à saisir le comportement attendu d'un système en cours de développement, sans avoir à préciser la façon dont ce comportement est réalisé.

Un cas d'utilisation ***regroupe une famille de scénarios d'utilisation***.

Il vise à mettre en évidence le cas particulier d'une utilisation du logiciel, répondant à un besoin du système. Si tous les cas particuliers d'utilisation du logiciel sont mis en évidence, les besoins de celui-ci ont donc été identifiés.

Fiche de description d'un cas d'utilisation (scénario)

Le diagramme des cas d'utilisation seul ne suffit pas à décrire les fonctionnalités du système à réaliser.

En complément on doit décrire, à l'aide d'une fiche descriptive en texte clair, les scénarios possibles lors du déroulement de chacun des cas d'utilisations présents sur le diagramme.

Cette fiche doit signaler le cas d'utilisation auquel elle se rattache, ainsi que sa version, son auteur, sa date de création, modification.

Un résumé du cas d'utilisation est aussi très utile.

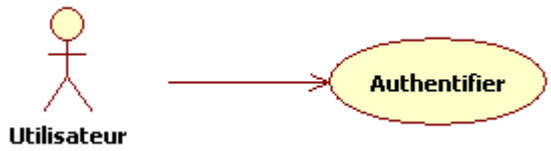
Puis les conditions nécessaires à l'entrée dans le cas d'utilisation, l'ensemble des scénarios, ainsi que

l'état du système après le déroulement normal du cas d'utilisation.

On distingue trois types de scénarios :

- ✔ Le ***scénario nominal*** : cas décrivant une utilisation normale de la fonctionnalité.
- ✔ Le ou les ***scénarios alternatifs***: cas optionnels ajoutés au scénario nominal ou erreurs récupérables lors du déroulement de ce scénario ou d'un autre scénario alternatif.
- ✔ Le ou les ***scénarios exceptionnels***: cas exceptionnels souvent fatals au bon déroulement de la fonctionnalité, pouvant arriver n'importe où (dans le scénario nominal ou ailleurs).

Exemple : scénario du cas d'utilisation « Authentifier »



Scénario nominal

Acteurs ou « Boundary »	Système
	N1 : Demande la saisie du « login » et du mot de passe.
N2 : L'utilisateur saisit son « login »et son mot de passe puis valide sa saisie.	
	N3 : Vérifie si le login et le mot de passe sont corrects.

Traitements alternatifs

Alternatif 1 : login inconnu

Début après le point N3

Acteurs ou « Boundary »	Système
	A1 : Indique que le login est inconnu

Retour au point N1

Alternatif 2 : mot de passe incorrect

Début après le point N3

Acteurs ou « Boundary »	Système
	A2 : Indique que le mot de passe est incorrect

Retour au point N1

Comme on vient de le voir sur cet exemple trivial, les scénarios se présentent sous la forme de tableaux où l'on place à droite le système lui-même et à gauche les acteurs ou éléments « frontière » du système (stéréotypé « Boundary » en UML).

S'en suit un jeu d'actions/ réactions de la part des protagonistes.

Les éléments « frontière » sont des équipements intelligents qui composent le système, mais sous forme d'entités autonomes (matériel comme une partie opérative ou un serveur annexe sur lequel le scénario ne s'exécute pas).

Dans le cas de notre médiathèque, les adhérents pourront principalement emprunter des médias, mais aussi consulter le catalogue en ligne, cela est même obligatoire pour notifier l'emprunt sur le site.

De plus, il est nécessaire d'authentifier l'adhérent emprunteur, mais aussi le gestionnaire qui valide

l'emprunt.

Le gestionnaire doit en plus gérer la médiathèque, c'est-à-dire l'ajout, la suppression et la modification des listes d'adhérents, mais aussi des médias.

D'où le diagramme des cas d'utilisation ci-dessous.

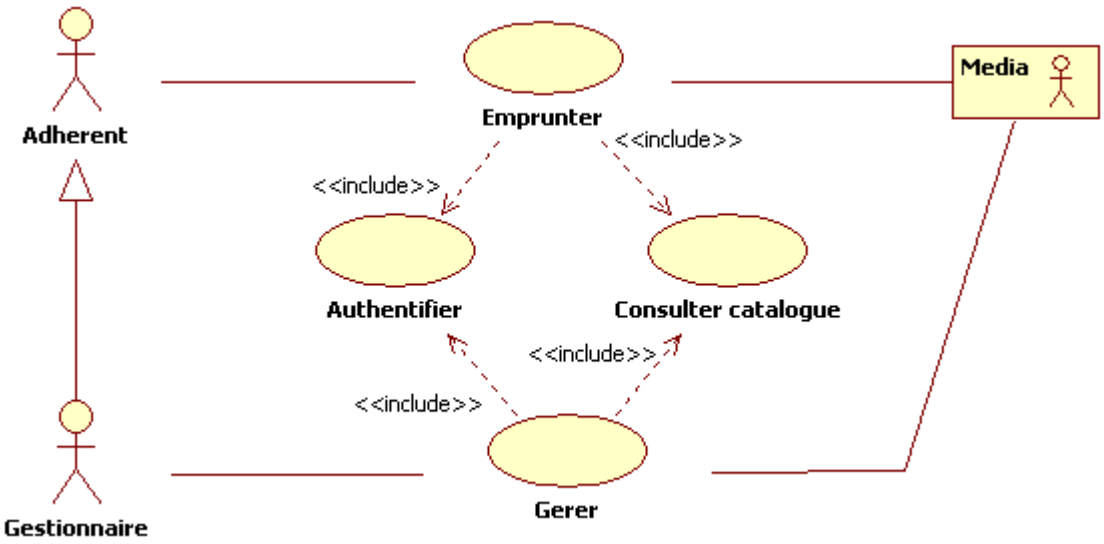


Diagramme des cas d'utilisations

On remarque sur le diagramme des cas d'utilisations que l'emprunt exige une consultation du catalogue (pour trouver la référence au média emprunté), ainsi qu'une authentification de l'emprunteur, ce qui se traduit par deux dépendances stéréotypées.

De même, le gestionnaire pour pouvoir gérer la médiathèque doit être dûment authentifié et a besoin aussi de consulter le catalogue. De plus, le cas d'utilisation « Emprunter » met en relation deux acteurs que sont l'adhérent et le média.

Par exemple, la dépendance entre les cas d'utilisations « Emprunter » et « Consulter catalogue » stéréotypée <<*include*>>, ce qui signifie que le cas d'utilisation « Emprunter » inclut l'exécution de « Consulter catalogue » dans son intégralité.

Remarque : Le deuxième stéréotype possible pour une dépendance de cas d'utilisation est <<*extends*>> qui signifie quand à lui qu'un cas d'utilisation étends les possibilités d'un autre. C'est à dire qu'il ajoute une fonctionnalité tout en conservant les fonctionnalités initiales. Cela ressemble à un héritage (qui est lui aussi possible), mais sans imposer la façon de l'implémenter, ce qui est préférable. Les autres stéréotypes disponibles avec certains AGL pour le diagramme des cas d'utilisations ne sont pas standard et on ne peut que déconseiller leurs utilisation.

Objets métiers

Les objets métiers sont le coeur de toute application, c'est eux qu'il faut absolument déterminer afin de rendre l'application conforme aux attentes du client, mais aussi afin de rendre la conception la plus souple possible pour les évolutions futures.

Les objets métiers ont ceci de particulier, c'est qu'ils sont persistants et leur cycle de vie est indépendant du cycle de vie de l'application qui les gèrent.

La recherche des objets métiers est basé sur l'analyse du cahier des charges, qui dans notre cas pourrait être résumé ainsi : « La médiathèque est géré par un ou plusieurs gestionnaires et permet à ses adhérents d'emprunter un certain nombre de médias de différent type pour une durée limitée moyennant le versement d'une cotisation ».

A la lecture de cette phrase, nous pouvons isoler ces objets métiers: des gestionnaires, des adhérents, des médias, des emprunts, des cotisations ...

Ils ont bien le caractère de persistance évoqué précédemment et ils sont bien à la source de toute l'activité d'une médiathèque. On pourrait être tenté de considérer la « cotisation » comme une propriété de l'adhérent, et lui attribuer par la même occasion le « versement de la cotisation ». Nous écartons cette solution, car nous considérons que ceci consisterait à attribuer à l'adhérent un comportement qui est propre à la logique métier. La logique métier est en effet davantage susceptible d'évoluer que l'objet métier en soi. On se doit de séparer ce qui est susceptible de varier (comme un comportement) de ce qui ne l'est pas (comme une caractéristique fondamentale).

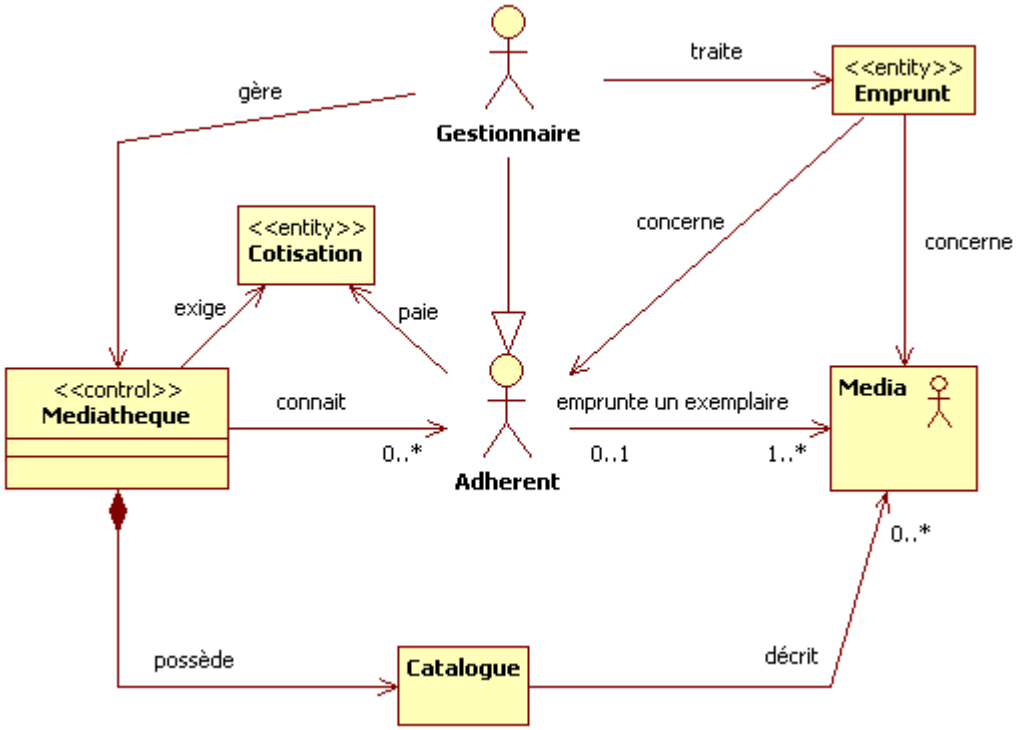
Les objets métiers sont modélisés en UML par des classes entités. Les acteurs du système sont aussi des classes entités pour notre application.

Les classes entités n'offrent aucun service, et ne contiennent que des informations qui les caractérisent et dont les instances sont persistantes. c'est à dire que lorsque l'application est arrêté, les données sont sauvegardées et lors de son futur redémarrage elle se retrouvera dans le même état.

Remarque : Une classe à la responsabilité de faire ***Une chose*** **et** ***une seule***, mais elle le fait bien ! Par exemple mettre une méthode « laver » dans une classe « Voiture » est une erreur, car ce n'est pas la Voiture qui ***se*** lave, mais la Station de lavage qui lave des Voitures.

Diagramme de classes d'analyse

Nous sommes maintenant en mesure de décrire avec le formalisme UML, notre application à l'aide d'un premier diagramme de classes. On trouve sur ce diagramme, les classes correspondantes pour les adhérents, les gestionnaires, les médias mais aussi les emprunts qui, s'ils ne sont pas des acteurs, n'en sont pas moins des objets entités qui représentent l'emprunt réel d'un média par un adhérent. A ce stade nous sommes encore en phase de spécification, le « quoi faire ». Mais son principal intérêt c'est qu'il a permis d'identifier les objets métiers et la façon dont ils interagissent. L'utiliser comme modèle de conception serait hasardeux. Cela entraînerait par exemple à confier à la classe entité « Cotisation » la responsabilité du paiement de la cotisation, ou pire, de confier à la classe « Media » la fonction d'enregistrement des emprunts.



Caractéristiques d'un adhérent

Un adhérent, c'est avant tout une personne qui a un nom et un prénom, qui normalement sont des caractéristiques qui ne peuvent changer. Son adresse par contre, elle le pourrait. Enfin, attribuer un numéro d'adhérent facilitera la gestion de la médiathèque.

Le gestionnaire, reprend l'ensemble des attributs d'un adhérent en rajoutant son login et son mot de passe.

Remarque : Par défaut, les attributs sont tous de visibilité privé.

Caractéristiques d'un média

Les caractéristiques d'un média sont: son type (CD, DVD, livre ...), le titre, l'auteur, un résumé, une catégorie, des mots clef de recherche et bien sûr un identifiant unique. D'autres informations sont sans doute nécessaires, par exemple un extrait (images, échantillons sonores ou vidéos ...). L'application étant de type web, il est possible associer à chaque média une page web descriptive, on ne conserve donc comme attribut pour cette classe une URL vers cette page, on pourra par la même occasion reporter le résumé dans cette page, ainsi que les mots clef qui pourront être pris en charge par une application d'indexation externe des pages web. Nous optons pour cette solution de façon à ne pas alourdir notre étude, notre propos étant davantage de présenter une méthodologie.

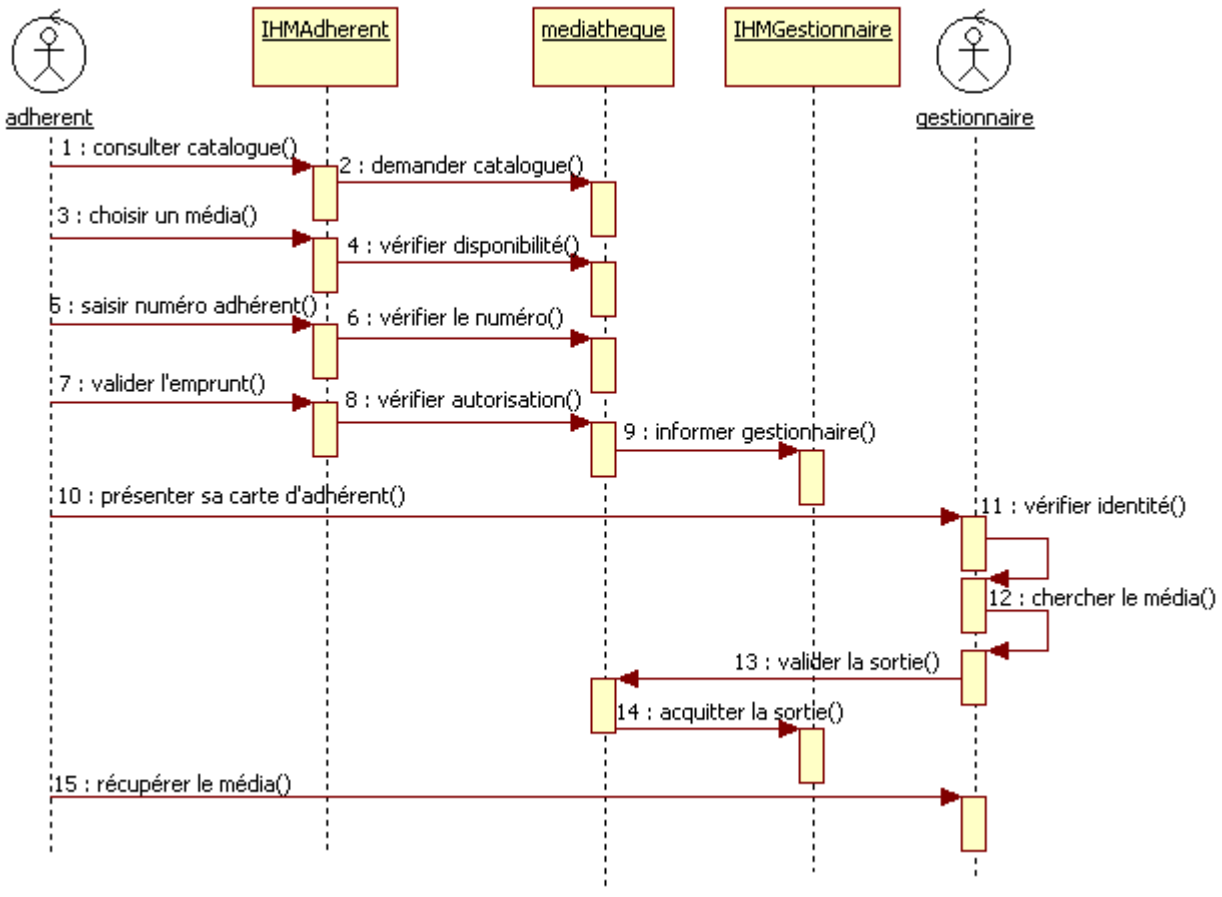
Caractéristiques d'un emprunt

Un emprunt est modélisé par une classe d'association. Cette classe met en relation un adhérent avec un certain nombre de médias empruntés. Le cahier des charges impose un maximum de 3 médias, mais on évite d'introduire cette contrainte au niveau de l'analyse, afin de faire face à une éventuelle modification. On préfèrera plutôt placer cette contrainte dans une classe de paramétrage.

Remarque : Un média peut être présent en plusieurs exemplaires dans la médiathèque, de ce fait on n'emprunte pas directement un média, mais un exemplaire de celui-ci.

Diagrammes de séquence

La réalisation chronologique d'un cas d'utilisation peut être décrite comme nous l'avons vu précédemment par un scénario, mais également en utilisant le formalisme UML grâce aux diagrammes de séquence. Chaque diagramme est relatif à un cas d'utilisation. A titre d'illustration, voici le diagramme de séquence du cas d'utilisation « Emprunter »:



IHMAdherent et IHMGestionnaire sont les interfaces homme/machine respectivement utilisée par l'adhérent et le gestionnaire, pour dialoguer avec le système.

Conception préliminaire

Construction du diagramme de classes de conception préliminaire

A ce stade, nous disposons d'un diagramme de classes d'analyse, et d'un ensemble de cas d'utilisation formalisé soit par des scénarios, soit par des diagrammes de séquences.

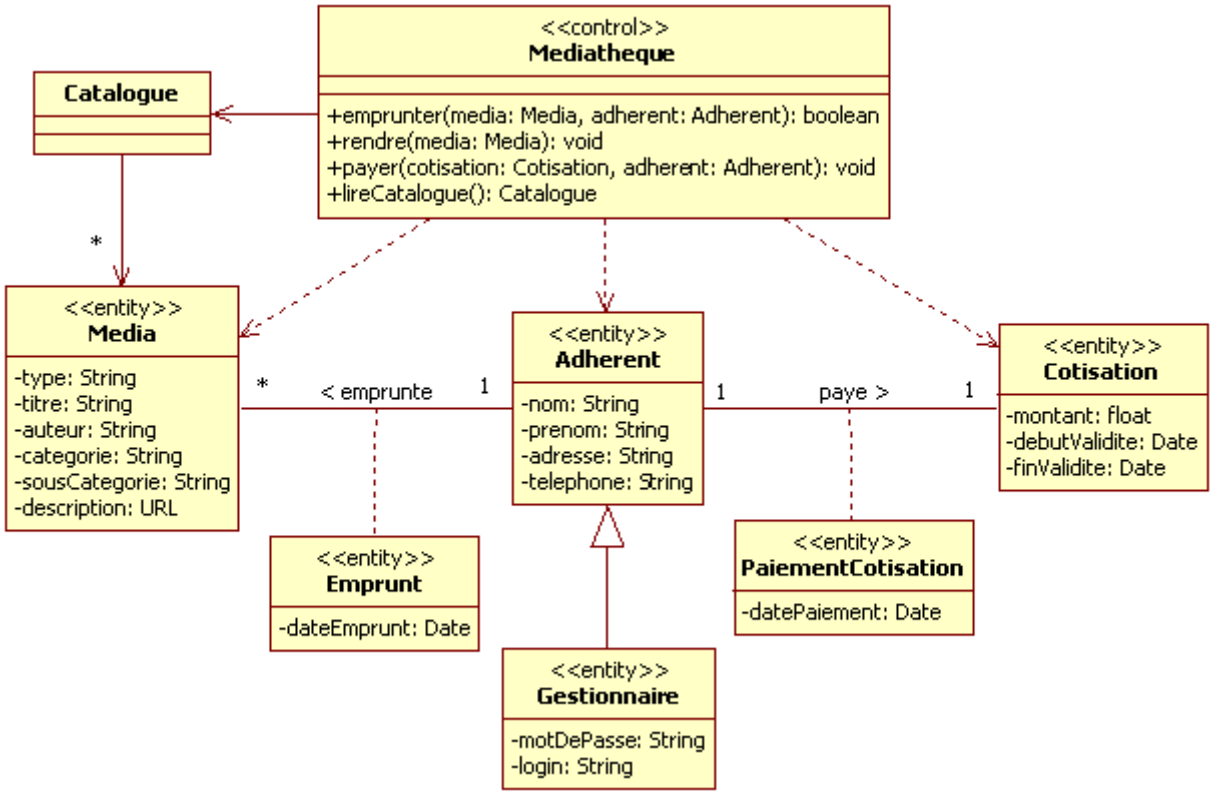
Les objets métiers ont été pour la plupart identifiés. Les attributs de ces objets ont été définis on utilisant les types génériques UML qui font abstraction de toute référence à un langage particulier.

Le passage à la phase de conception, nous oblige à faire des choix technologiques et notamment de préciser le choix du langage, par exemple le type « string » UML pourra être implémenté par un « char* » en C++ ou mieux un « string » STL, ou par un String en java. La plupart des outils UML étant capable de générer le squelette des classes (et même de faire du reverse), il est important de se conformer au langage dans lequel on implémente.

L'adoption de « règles de codage », par le respect d'une standardisation du nommage des classes et de leurs méthodes, s'avèrent aussi indispensable pour améliorer la lisibilité et la maintenabilité du code. Ceci permet de s'inscrire dans une démarche qualité (ISO9000).

Diagramme de conception préliminaire

Notre voici est présence de notre premier diagramme de conception qui fait clairement apparaître les objets métiers et les liens qui les unissent. Beaucoup de chemin reste encore à parcourir, avant d'arriver à une solution répondant au cahier des charges. Le fonctionnel est pour l'instant globalement confié à la classe « Mediatheque », où figure certain des services à assurer qui ont été identifiés à ce stade de la conception.



Les classes d'association

Nous avons deux classes d'association sur notre diagramme de classes conceptuel : les classes « PaiementCotisation » et « Emprunt ».

Une classe d'association, est une classe qui permet d'associer deux classes qui ne le sont pas directement. Plus précisément, elle consiste à représenter une association également par une classe. Quand avons-nous besoin d'utiliser une classe d'association ? Et bien, à chaque fois que l'on a besoin de caractériser l'association elle-même, autrement dit si on veut définir des attributs pour l'association. Prenons le cas de l'association entre un adhérent et une cotisation, celle-ci peut être décrite par la phrase « un adhérent paye une cotisation ». Un adhérent est modélisé par la classe Adherent dont l'attribut « nom » représente le nom de l'adhérent, de même que l'attribut « montant » de la classe « Cotisation » représente le montant de la cotisation dont il doit s'acquitter. Mais alors si on veut conserver une trace de la date de paiement de cette cotisation, où doit-on placer cette information ? Dans la classe Adherent ? dans la classe Cotisation ? ou dans les deux ? Aucune de ces solutions n'est satisfaisante, car en définitive la « date de paiement » ne caractérise ni l'adhérent ni la cotisation, c'est en fait une caractéristique de l'association elle-même.

Ainsi, les classes Adherent et Cotisation ne sont pas associées directement, mais par une classe intermédiaire (dite classe d'association). L'acte de paiement de la cotisation, modélisé par la classe « PaimentCotisation » concerne un Adhérent et une Cotisation, celle-ci possède un attribut qui est la date de paiement, et elle doit aussi, bien entendu, posséder une référence sur chacun d'eux.

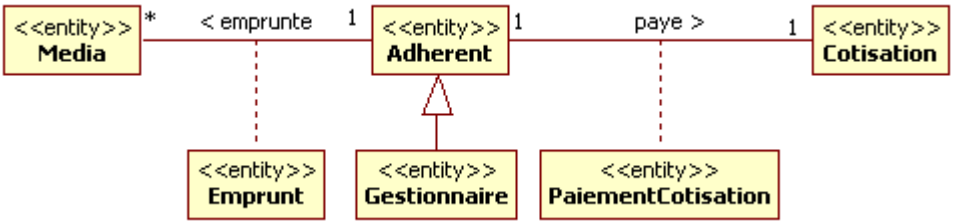
Conclusion provisoire

Notre travail est loin d'être terminé, dans la prochaine étape nous aborderons la conception détaillée de notre application, ceci nous amènera à compléter et sans doute à améliorer ou même à corriger notre modèle (au vu du diagramme, nous voyons déjà que la notion d'exemplaire, que nous avions signalé dans la spécification, n'apparaît pas, il est donc nécessaire de perfectionner notre modèle) . La modélisation UML est un processus itératif, tout au long duquel la recherche de la solution définitive ne peut se faire que par étapes successives.

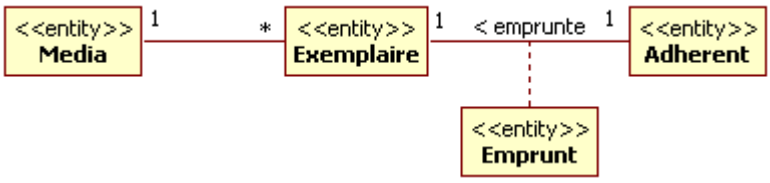
Deuxième partie - Conception avancée & Modèle 3-tier

Cette série d'articles concernent l'analyse et la conception objet, à partir d'une étude de cas qui est celle de la gestion d'une médiathèque. Dans notre précédent article, nous avons montré comment utiliser le formalisme UML pour spécifier et analyser notre système en suivant un cycle de développement en V. Puis nous avons commencé une phase de conception, dite conception préliminaire, dont le rôle principal a été de trouver les objets métiers, et à partir de laquelle nous allons poursuivre notre étude.

Pour les lecteurs qui n'ont pas eu la possibilité de lire le précédent article (dont, bien entendu, nous recommandons la lecture), nous rappelons brièvement notre cahier des charges : « La médiathèque est géré par un ou plusieurs gestionnaires et permet à ses adhérents d'emprunter un certain nombre de médias de différent type pour une durée limitée moyennant le versement d'une cotisation ». Notre étude nous a conduit à l'élaboration d'un premier diagramme de classes montrant les « classes métiers ».



L'association entre adhérent et média, que l'on peut traduire ainsi: « un adhérent emprunte plusieurs médias », n'est pas satisfaisante. Nous avions déjà soulevé ce point, en précisant que notre cahier des charges prévoit que la médiathèque puisse disposer de plusieurs exemplaires d'un même média, et par conséquent, il nous faut modéliser un « exemplaire », et modifier cette association. En effet un adhérent emprunte un exemplaire d'un média, et non le média en soi. Il sera également nécessaire de s'assurer qu'un adhérent n'emprunte pas plusieurs exemplaires du même média, mais ceci est une « règle métier » et non une caractéristique d'une classe particulière (nous verrons ultérieurement où et comment cette contrainte sera intégrée). Notre association est ainsi modifiée, en faisant apparaître la classe « Exemplaire »:



Règles de conception

Nous allons à présent aborder la conception détaillée de notre application.

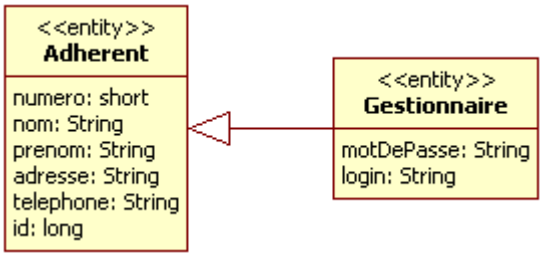
Mais tout d'abord, nous allons définir un certain nombre de règles de construction de notre modèle conceptuel, que nous utiliserons tout au long de notre conception et qui nous permettra aussi la découverte des classes dites de service. Certaines de ces règles nous ont d'ailleurs implicitement déjà servi pour la découverte des objets métiers et persistants.

Règle de construction conceptuelle 1 :

Les objets métiers, gérés par le système, sont persistants et chaque catégorie est représenté par une classe « entité » qui lui correspond. Les acteurs, humain et non humain dont le système doit conserver les informations rentrent aussi dans cette catégorie .

On trouve donc une classe entité pour les médias, mais aussi pour les gestionnaires, les adhérents ...

Le nom de la classe est celui de l'objet métier, repris du cahier des charges.



Les classes « entités » Gestionnaire et Adherent

En UML, une classe de stéreotype "entity" représente une entité faisant partie du système, c'est-à-dire un objet réel existant dans le système ou une information. Elle n'assure aucune action de contrôle sur le système (aucun traitement), elle stocke simplement des informations qui caractérisent l'objet réel représenté ou son comportement.

Règle de construction conceptuelle 2 :

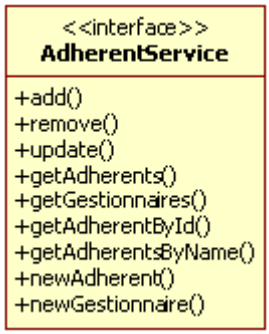
Chaque information échangée, liée à l'activité du système et au contexte dynamique, et dont il faut assurer la persistance, est également représenté par une classe « entité ».

Les activités liées à l'utilisation de la médiathèque, qui consiste à emprunter ou payer une cotisation, doivent aussi être modélisées par des classes entités, car il faut rendre persistantes les informations qu'elles véhiculent. Comme par exemple la réalisation d'un emprunt ou le paiement d'une cotisation via l'échange d'une somme d'argent contre une carte d'adhérent. On trouve ainsi, les classes Emprunt, Cotisation ...

Règle de construction conceptuelle 3:

Chaque classe « entité » est gérée par une classe de service, destiné à manipuler l'objet en respectant des règles métiers. La classe de service est en fait une « interface » au sens UML, qui offre au minimum les services d'ajout et de création, de suppression, de récupération d'un ou d'un ensemble d'éléments (via des critères spécifiques) et la modification d'un élément existant.

Le nom de ces interfaces est celui de la classe entité suivie du mot Service. Par exemple pour la classe de service de la classe entité Adherent l'interface s'appellera AdherentService.



Une interface de service et le code Java associé.

```
package entreprise;

import java.util.List;

public interface AdherentService {
  void add(Adherent a) throws EntrepriseException;
  void remove(Adherent a) throws EntrepriseException;
  void update(Adherent a) throws EntrepriseException;
  List<Adherent> getAdherents() throws EntrepriseException;
  Adherent getAdherentById(long id) throws EntrepriseException;
  List<Adherent> getAdherentsByName(String name) throws EntrepriseException;
  List <Gestionnaire> getGestionnaires() throws EntrepriseException;
  Adherent newAdherent(String nom, String prenom, String adr, String tel)
       throws  EntrepriseException;
  Gestionnaire newGestionnaire(Adherent a, String login, String motDePasse)
       throws EntrepriseException;
}
```

Remarques :

1. Le concept d'interface est natif en Java, mais pas en C++. Pour rappel une interface ne contient que des définitions de méthodes (elles sont toutes abstraites) et ne possède aucun attribut (Java déroge en parti à cette règle). En C++ on utilisera donc une classe abstraite où toutes les méthodes sont virtuelles pures.

2. L'acteur Gestionnaire étant une sorte d'Adherent, il partage la même classe de service, celle de la classe acteur ancêtre : AdherentService

Règle de construction conceptuelle 4:

Chaque « réalisation » d'un cas d'utilisation est modélisé par une classe (ou paquetage) dite de contrôle, et ayant le stéréotype « control ». Cette classe intègre le comportement dynamique de l'application et plus précisément d'un cas d'utilisation particulier.

Ici pour l'application qui tourne sur le serveur, la classe principale de contrôle est la classe Médiathèque. Selon la complexité du traitement à réaliser, la classe pourra être remplacée par un ensemble de classes qui collaborent entre elles et que l'on regroupera dans un même paquetage.

Règle de construction conceptuelle 5:

Les dispositifs utilisés par le système, mais qui n'en font pas réellement partie sont modélisés par des classes stéréotypée « boundary ».

Une classe de stéreotype "boundary" représente un bord du système, c'est-à-dire une classe à la frontière du système qui sert à échanger de l'information avec une entité extérieure au système. Exemple : partie opérative, capteurs ...

Ces classes gèrent en général l'accès au « matériel », elles ont généralement une instance unique dans le logiciel (un objet pour gérer une carte ou un accès à la base de données, et non plusieurs objets pour cette ressource qui est physiquement unique), on utilise de ce fait le motif « Singleton ».

La règle de nommage de ces interfaces, est la même que celle déjà vu à la règle 3.

Dans notre application médiathèque, nous n'avons pas ce genre de classe. Mais cela peut se présenter, si on ajoute à notre système un lecteur optique pour lire automatiquement le code à barres du média au moment de l'emprunt.

Règle de construction conceptuelle 6 :

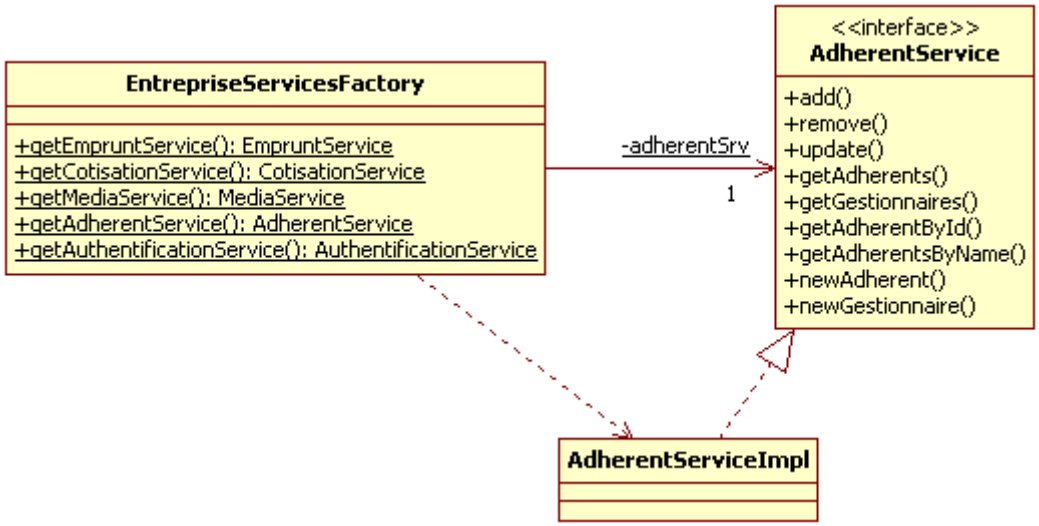
Le choix d'une implémentation / réalisation d'une interface de service se fait via le motif de conception fabrique d'objets (« Factory »).

Les services métiers étant définis par des interfaces, l'accès à une implémentation se fait au travers de l'utilisation d'une fabrique d'objets, par le biais de l'utilisation du motif de conception « Factory ».

Le motif « factory » permet surtout de cacher l'implémentation des services vis à vis de l'utilisateur. En java, on pourra favorablement utiliser pour cela le mécanisme de protection lié à la visibilité de type paquetage pour cacher l'implémentation.

Ainsi, seule la classe de contrôle qui gère le cas d'utilisation ainsi que, dans un premier temps, les classes entités seront publiques.

La factory s'appellera <Nom du package>ServicesFactory.



Le motif Factory appliqué à l'interface AdherentService et sa méthode « getAdherentService() »

Portion de code de la classe EntrepriseServicesFactory:

```
package entreprise;

public class EntrepriseServicesFactory {

  private static EmpruntService empruntSrv;

  synchronized public static EmpruntService getEmpruntService() {

        if(empruntSrv == null)
            empruntSrv = new EmpruntServiceImpl();

        return empruntSrv;
   }
   ...
}
```

Pourquoi synchronized ? Si plusieurs processus (thread), accède simultanément à la méthode getXX avant la création effective de l'instance, le singleton ne fonctionne pas (risque de création de plusieurs instances).

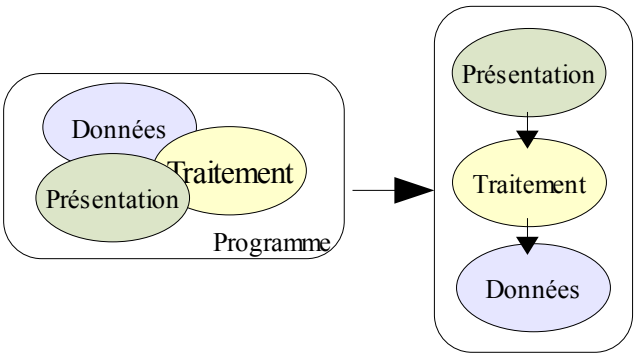
Conception détaillée

Dans cette phase, nous allons compléter nos diagrammes de classes de conception en partant de notre analyse et de notre conception préliminaire, ou nous avons détaillé l'ensemble des objets métiers.

Nous allons vous présenter la méthode de conception dite n-tier, dans sa version simplifiée à trois couches (tier veut dire couche en anglais US).

Conception n-tier

Tout programme informatique présente le résultat d'un traitement effectué sur des données. Il est important que dans le processus de développement, ces trois entités présentation-traitement-données soit clairement séparées.

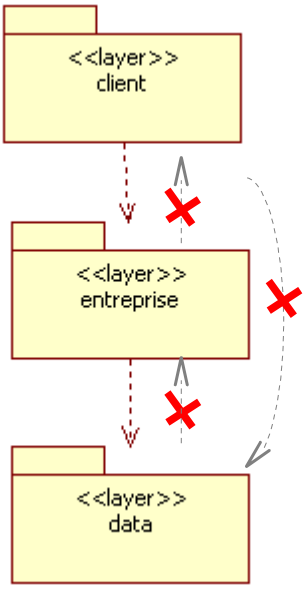


La conception n-tier (ou conception en couches) se base sur cette séparation et s'inscrit dans une démarche de qualité du logiciel dont les critères sont:

- la maintenabilité
- la lisibilité
- l'évolutivité
- la réutilisabilité
- la testabilité
- la facilité de développement en équipe

Une fois les couches bien identifiées, le couplage entre celles-ci doit être le plus faible possible.

Architecture 3-tier



Règles imposées par l'architecture

- Une couche dépend uniquement de la couche immédiatement inférieure
- Une modification dans une couche ne doit avoir aucune influence sur la couche supérieure
- La dépendance est unidirectionnelle et est toujours dirigé d'une couche supérieure vers la couche immédiatement inférieure

Description de la couche client

La couche client s'occupe de la « présentation ». Elle regroupe toutes les interfaces homme-machine (IHM) offertes à l'utilisateur. Elles peut contenir plusieurs applications qui s'appuient sur les mêmes règles métiers et le même système d'informations. Les applications offertes peuvent être de différentes type (console (CUI), graphique (GUI) ou Web).

Les classes dans cette couche sont liées à la bibliothèque graphique utilisée, pour les applications GUI autonome.

Description de la couche entreprise

La couche entreprise est la couche principale, appelée couche métier, car c'est elle qui contient l'ensemble des objets et services métiers.

On y retrouvent l'ensemble des classes entité (les objets métiers), ainsi que les classes de services, (ses services métiers) c'est à dire les interfaces avec au moins une implémentation, vu précédemment.

On y trouve aussi la logique transactionnelle (mise à jour des données) et la logique fonctionnelle, qui peut être transféré dans une couche supérieure si l'application est trop complexe dans une version à cinq couches par exemple.

Description de la couche physique

Cette couche comprend une ou deux sous couches, une pour l'accès au donnée (sous couche « data » qui assure la persistance des données et le mappage relationnel/objet) et l'autre pour l'accès au matériel spécifiques (carte IO, port RS232, etc..) dans la sous couche « io ».

Son rôle est de limiter la dépendance du logiciel vis à vis du matériel, et du mécanisme de persistance des données.

Possibilités offertes par l'architecture 3-tier

La passage d'une application autonome GUI vers une application Web est facilité, les couches entreprise et physique sont conservées, les modification étant concentré dans la seule couche client.

Le remplacement d'une sauvegarde sur fichier des données persistantes, par une sauvegarde dans une base de données, ou un changement de base de données est aussi facilité. Seule la couche physique est concernée, les autre n'étant pas modifiées. Le choix d'une implémentation dans une couche étant réaliser dans la fabrique d'objet de la couche elle-même.

La conception est plus claire et plus modulaire.

De plus, le développement est facilité, chaque couche est développé indépendamment des autres.

L'intégration sans trouve grandement simplifiée.

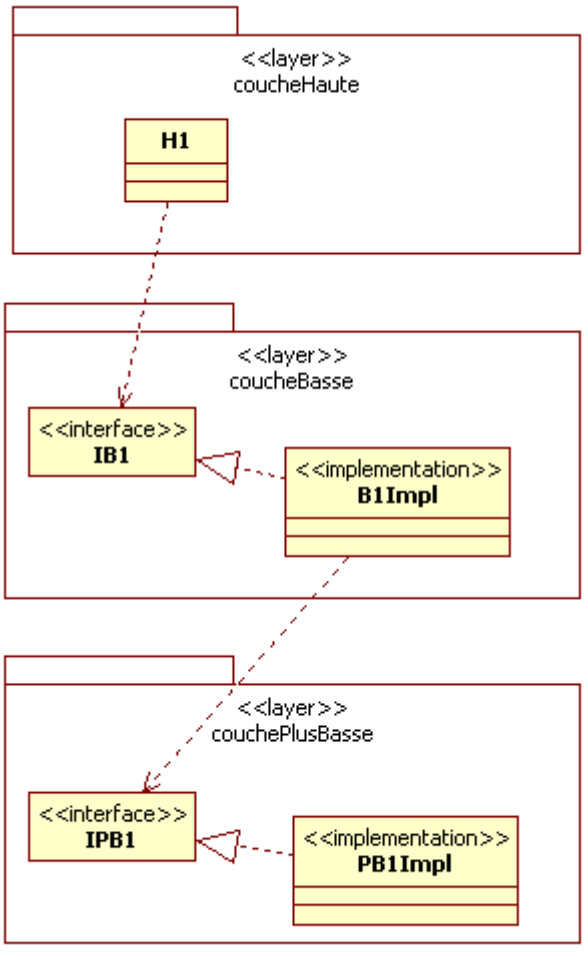
Assurer l'indépendance entre couches

- H1 utilise l'interface IB1
- Une modification de B1Impl n'a aucun effet sur H1
- B1Impl accède à la couche plus basse en utilisant l'interface IPB1, toute modification de l'implémentation PB1Impl n'a aucun effet sur la « coucheBasse », ni sur la « coucheHaute »

Règle

Une couche supérieure ne doit accéder qu'aux **interfaces** de la couche inférieure

Ainsi, toutes les classes d'implémentation doivent avoir une **visibilité de type package**, pour en empêcher l'accès depuis la couche supérieure.



Question

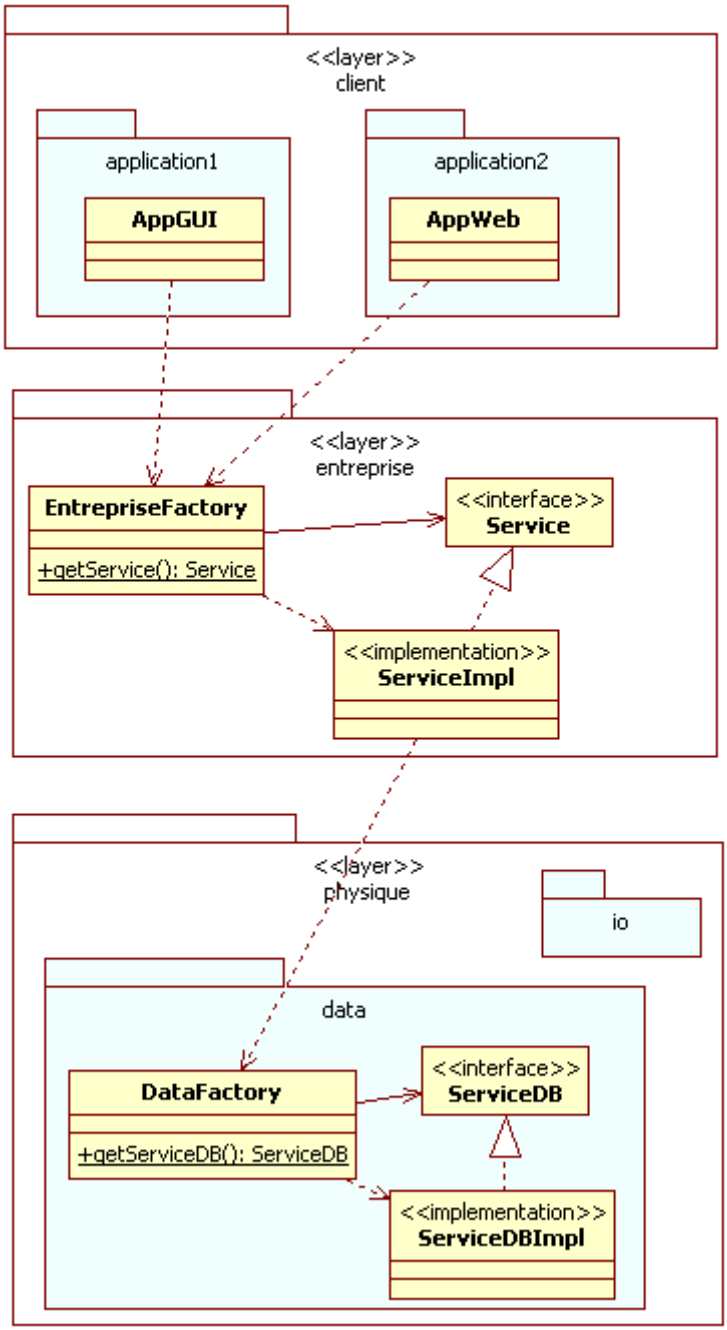
Comment la couche supérieure peut-elle utiliser une interface qui par définition ne peut être instancié ?

Réponse

Grâce au motif Factory

Une couche supérieure n'a qu'un seul point d'entrée dans la couche inférieure: la Factory

EntrepriseFactory fournit le service « **Service** », ce dernier (plus précisément son implémentation) obtient auprès de **DataFactory** le service de plus bas niveau « **ServiceDB** ».



La gestion des exceptions

En phase de spécification des services assurés par une interface, on ne sait pas quel en sera l'implémentation et ceci n'est à priori pas en prendre en considération à ce niveau. Cependant lors de l'implémentation, des exceptions peuvent être générées et celles-ci ne peuvent pas être prévues à l'avance, car elles sont précisément dépendantes de l'implémentation qui sera faite ultérieurement.

De plus, selon l'implémentation retenue, une exception peut être générée ou non. Si une méthode de l'interface a été spécifiée comme ne générant pas d'exception, une évolution de l'application, entraînant une nouvelle implémentation susceptible de générer une exception, nous conduit à deux alternatives.

- La nouvelle implémentation capture l'exception et la traite localement, ceci est par principe un mauvais choix, car l'application cliente n'est jamais informée des dysfonctionnements (exemple d'une connexion réseau défectueuse, la couche basse capture et traite l'exception mais n'en informe pas la couche supérieure car elle n'en a pas le moyen),

- ou bien il faut redéfinir la signature de la méthode de l'interface pour indiquer qu'elle peut générer une exception, ce qui impacte la couche client, et toutes les applications concernées, et oblige à toutes les modifier, et justement ceci est contraire au principe de séparation des couches.

On s'impose donc la règle de conception suivante:

Toutes les méthodes d'une interface sont à priori susceptibles de générer une exception

Mais alors comment spécifier l'exception, puisqu'elle est dépendante de l'implémentation ? Utiliser l'exception générique « Exception » dont hérite toutes les autres ?

Un choix plus judicieux, consiste à définir pour chaque couche, sa propre classe d'exception en dérivant de la classe « Exception ». Cette dernière prévoit toutes les fonctionnalités nécessaires pour encapsuler l'exception d'origine (voir méthode getCause() de la classe de base).

Ainsi, dans la couche « entreprise » nous définissons une classe « EntrepriseException » qui dérive de la classe « Exception ». Toutes les méthodes des interfaces de service sont susceptibles de générer cette exception. Occasionnellement, il est possible d'exclure de cette règle certaines méthodes, par exemple une méthode qui est une méthode purement utilitaire (trier une liste, extraire une liste d'une autre liste ...), mais là encore la vigilance est de rigueur.

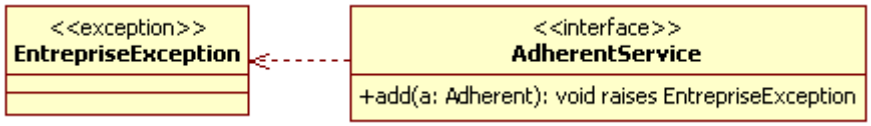
Il est important que les classes d'implémentation remontent toujours les exceptions et ne les consomment pas localement. Ceci est également une règle importante de conception:

Les implémentations des interfaces ne doivent pas consommer les exceptions, mais au contraire doivent toujours les remonter en encapsulant la cause initiale dans une exception générique propre à la couche.

Modéliser une exception en UML

En UML, il est possible de spécifier qu'une opération dans une classe peut générer un événement particulier. UML utilise un signal (classe avec le stéréotype <<signal>>) pour spécifier un événement. On spécifie ensuite que tel opération dans une classe donnée, peut générer ce signal. UML utilise le stéréotype <<exception>> pour désigner un signal particulier qui s'apparente davantage à une erreur ou un dysfonctionnement. Une exception UML est donc un cas particulier de signal.

Ainsi, sur le schéma ci-dessous, on spécifie que la méthode *add()* de l'interface « AdherentService » est susceptible de générer l'exception « EntrepriseException ».



Exemple de code:

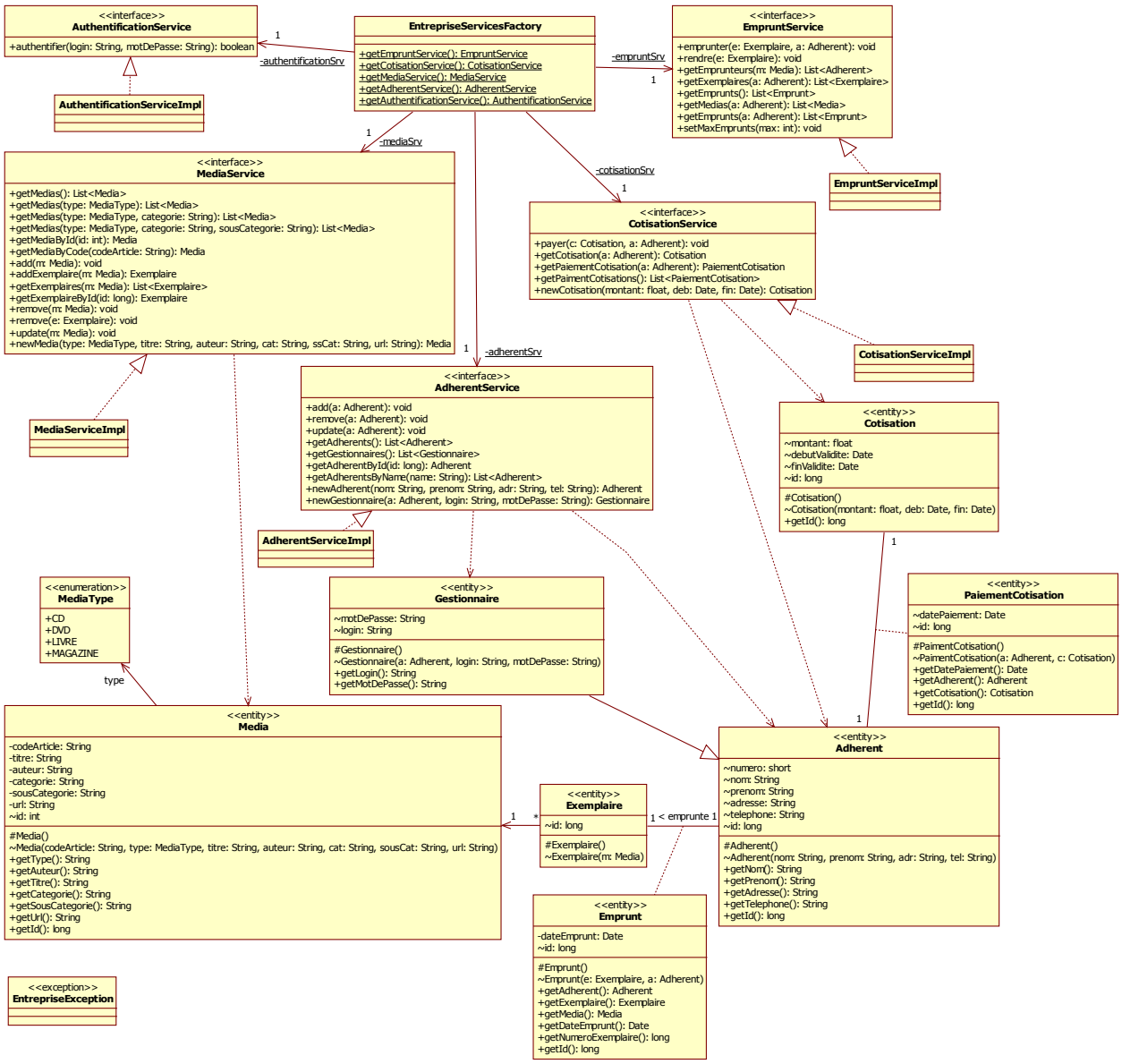
```
public void add(Adherent a) throws EntrepriseException {

  try {
    // invoque la couche data, susceptible de générer l'exception DataException propre à la couche

  } catch (DataException ex) {
      throw new EntrepriseException(ex);
  }
}
```

Le package « entreprise »

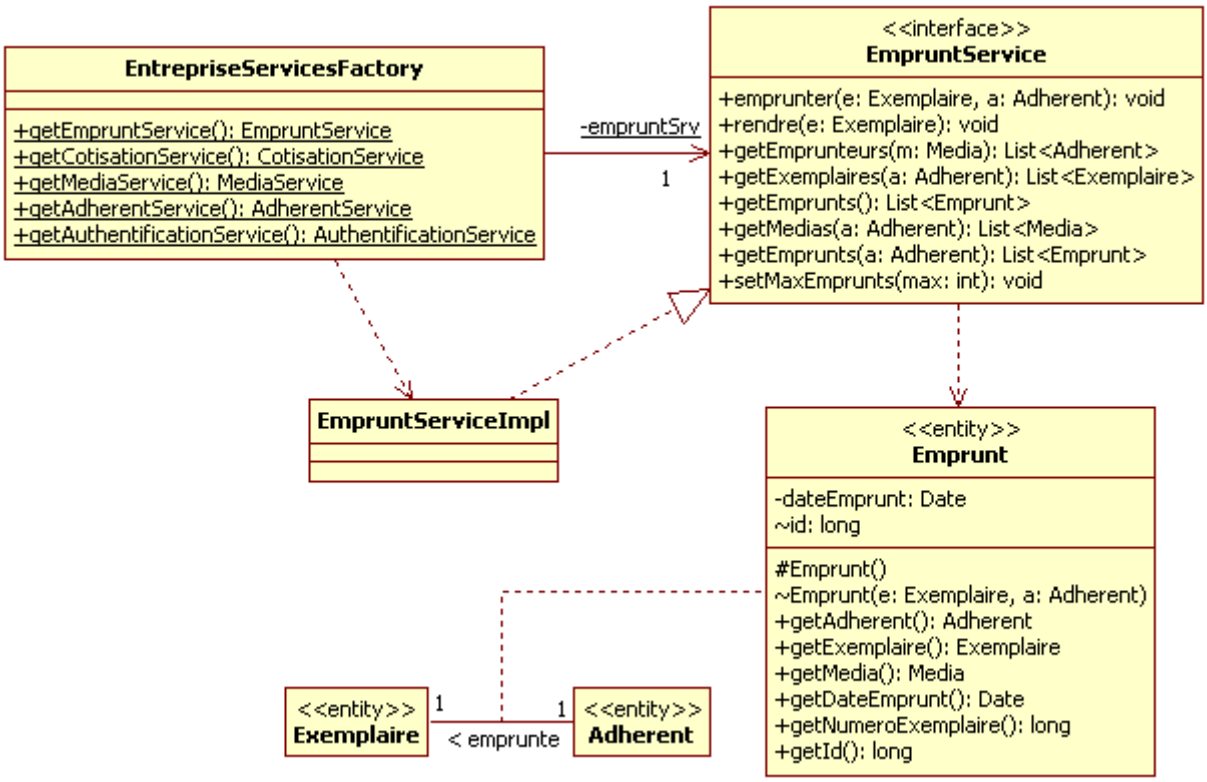
La modélisation de la couche « entreprise » est la première étape à réaliser. Nous nous basons essentiellement sur les règles de conception de bases que nous avons énoncées (voir l'article

précédent). Dans cette couche il s'agit surtout de modéliser toutes les entités (objets métiers), les relations qui les unissent (OneToOne, OneToMany, ManyToOne), et les services qui les gèrent. Ces services gèrent la partie fonctionnelle de notre application, et le respect des règles métiers. Tous les services liées au transactionnel seront déléguées à la couche inférieure.

Le diagramme de classes du package entreprise



Détail du service Emprunt



Dans le contexte dynamique , l'action d'emprunter crée un objet « Emprunt » qui doit être rendu persistant. L'objet Emprunt encapsule toutes les propriétés propres à l'emprunt, à savoir l'adhérent emprunteur, l'exemplaire du média emprunté ainsi que la date d'emprunt (nous avons vu qu'en UML, ceci est modélisé en utilisant une classe d'association).

Il n'est pas de la responsabilité de cet objet de vérifier si l'adhérent est autorisé à faire cet emprunt et les conditions liées à cet emprunt. Tout ceci est délégué à la classe de service associé à la gestion de cet objet, en l'occurrence ici, il s'agit de la classe « EmpruntService ». C'est elle qui applique les règles métiers afin de valider ou non l'emprunt. Dans notre cas, ces règles métiers peuvent s'énoncer comme suit:

- l'adhérent doit être à jour de sa cotisation
- il ne peut emprunter qu'un nombre limité de médiathèque
- il ne peut emprunter qu'un seul exemplaire du même média

Si les règles de gestion de la médiathèque changent ou évoluent, les objets entités de type Emprunt, ne sont nullement affectés. Seul l'implémentation du service concerné doit être redéfini. Nous voyons ainsi, comment le modèle global proposé, peut répondre au critère qualité notamment d'évolutivité.

Voici à titre d'exemple, le code java de la classe **EmpruntServiceImpl**.

```
package entreprise;

import data.DataException;
import data.DataService;
import data.DataServicesFactory;
import java.util.Date;
import java.util.Iterator;
import java.util.List;
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;

class EmpruntServiceImpl implements EmpruntService {

    DataService dataSrv;
    int maxEmprunt = 3;

    EmpruntServiceImpl() {
```

```
        this.dataSrv = DataServicesFactory.getDataService();
    }

    private void verifier(Media m, Adherent a) throws EntrepriseException, DataException {
        // vérifier si l'adherent a cotisé
        CotisationService cotisationSrv = EntrepriseServicesFactory.getCotisationService();
        PaiementCotisation paiement = cotisationSrv.getPaiementCotisation(a);
        if (paiement == null || new Date().after(paiement.getCotisation().finValidite)) {
            throw new EntrepriseException("L'emprunt est impossible en raison d'un défaut de
cotisation.");
        }

        // vérifier si l'adhérent peut encore emprunter
        List<Emprunt> list = this.dataSrv.getEmprunts(a);
        if (list.size() >= maxEmprunt) {
            throw new EntrepriseException("Le nombre d'emprunts est limité à " + maxEmprunt + ".");
        }

        for (Emprunt ep : list) {
            if (ep.exemplaire.media.equals(m)) {
                throw new EntrepriseException("On ne peut emprunter qu'un exemplaire du même
média.");
            }
        }
    }

    public void emprunter(Exemplaire e, Adherent a) throws EntrepriseException {
        try {
            Exemplaire exp = null;
            Media m = e.media;
            this.verifier(m, a);
            List<Exemplaire> listExpl = this.getExemplairesDisponible(m);
            if (!listExpl.contains(e)) {
                throw new EntrepriseException("L'exemplaire n'est pas disponible.");
            }
            Emprunt ep = new Emprunt(e, a);
            this.dataSrv.add(ep); // rendre persistant
        } catch (DataException ex) {
            Logger.getLogger(EmpruntServiceImpl.class.getName()).log(Level.SEVERE, null, ex);
            throw new EntrepriseException(ex);
        }
    }

    public List<Exemplaire> getExemplairesDisponible(Media m) throws EntrepriseException {
        try {
            List<Emprunt> listEmp = this.dataSrv.getEmprunts(m);
            List<Exemplaire> listExp = this.dataSrv.getExemplaires(m);
            for (Emprunt ep : listEmp) {
                for (Iterator<Exemplaire> it = listExp.iterator(); it.hasNext();) {
                    if (it.next().equals(ep.exemplaire)) {
                        it.remove();
                        break;
                    }
                }
            }
            return listExp;
        } catch (DataException ex) {
            Logger.getLogger(EmpruntServiceImpl.class.getName()).log(Level.SEVERE, null, ex);
            throw new EntrepriseException(ex);
        }
    }

    public List<Exemplaire>[] getExemplairesEmprunteEtDisponible(Media m) throws
EntrepriseException {
        try {
            List<Emprunt> listEmp = this.dataSrv.getEmprunts(m);
            List<Exemplaire> listExp = this.dataSrv.getExemplaires(m);
            List<Exemplaire> listExp2 = new Vector();
            for (Emprunt ep : listEmp) {
                listExp2.add(ep.exemplaire);
                for (Iterator<Exemplaire> it = listExp.iterator(); it.hasNext();) {
                    if (it.next().equals(ep.exemplaire)) {
                        it.remove();
                        break;
                    }
                }
            }
            List<Exemplaire>[] ret = new Vector[2];
            ret[0] = listExp2;
            ret[1] = listExp;
            return ret;
        } catch (DataException ex) {
            Logger.getLogger(EmpruntServiceImpl.class.getName()).log(Level.SEVERE, null, ex);
            throw new EntrepriseException(ex);
        }
    }
```

```
    public void rendre(Exemplaire exp) throws EntrepriseException {
        try {
            Emprunt e = this.dataSrv.getEmprunt(exp);
            this.dataSrv.remove(e);
        } catch (DataException ex) {
            Logger.getLogger(EmpruntServiceImpl.class.getName()).log(Level.SEVERE, null, ex);
            throw new EntrepriseException(ex);
        }
    }

    public List<Exemplaire> getExemplaires(Adherent a) throws EntrepriseException {
        try {
            List<Exemplaire> list = new Vector();
            List<Emprunt> eps = this.dataSrv.getEmprunts(a);
            for (Emprunt ep : eps) {
                list.add(ep.exemplaire);
            }
            return list;
        } catch (DataException ex) {
            Logger.getLogger(EmpruntServiceImpl.class.getName()).log(Level.SEVERE, null, ex);
            throw new EntrepriseException(ex);
        }
    }

    public List<Emprunt> getEmprunts(Adherent a) throws EntrepriseException {
        try {
            return this.dataSrv.getEmprunts(a);
        } catch (DataException ex) {
            Logger.getLogger(EmpruntServiceImpl.class.getName()).log(Level.SEVERE, null, ex);
            throw new EntrepriseException(ex);
        }
    }

    public List<Emprunt> getEmprunts() throws EntrepriseException {
        try {
            return this.dataSrv.getAll(Emprunt.class);
        } catch (DataException ex) {
            Logger.getLogger(EmpruntServiceImpl.class.getName()).log(Level.SEVERE, null, ex);
            throw new EntrepriseException(ex);
        }
    }

    public List<Emprunt> getEmprunts(Media m) throws EntrepriseException {
        try {
            return this.dataSrv.getEmprunts(m);
        } catch (DataException ex) {
            Logger.getLogger(EmpruntServiceImpl.class.getName()).log(Level.SEVERE, null, ex);
            throw new EntrepriseException(ex);
        }
    }

    public List<Adherent> getEmprunteurs(Media m) throws EntrepriseException {
        try {
            List<Adherent> list = new Vector();
            List<Emprunt> eps = this.dataSrv.getEmprunts(m);
            for (Emprunt ep : eps) {
                list.add(ep.adherent);
            }
            return list;
        } catch (DataException ex) {
            Logger.getLogger(EmpruntServiceImpl.class.getName()).log(Level.SEVERE, null, ex);
            throw new EntrepriseException(ex);
        }
    }

    public void setMaxEmprunts(int max) {
        if(max < 1)
            throw new IllegalArgumentException();
        this.maxEmprunt = max;
    }
}
```

Le package « data »

Cette couche gère la persistance des données et le transactionnel.

Rendre les objets persistants, consiste à les sauvegarder sur un support de persistance, en pratique dans une base de données. L'utilisation d'une base de données objet peut faciliter cette tâche. Mais pour des raisons de performances, l'utilisation de base de données relationnelle est plus répandue.

La manière dont les informations sont encapsulées dans un objet et celle utilisée dans les SGBDR est intrinsèquement différente. Un SGBDR stocke des valeurs primaires (entier, chaînes ...) dans des

tables.

Un objet peut posséder des attributs de type primaires mais aussi de type objet. Les notions d'héritage et de polymorphisme, propre à la programmation objet n'ont pas d'équivalent dans les SGBDR. La couche de persistance doit donc se charger de cette conversion, que l'on appelle communément le « mappage relationnel/objet ».

Le rôle de la couche « data » est de rendre la couche « entreprise » totalement indépendante de la manière dont les objets sont sauvegardés puis récupérés à partir du support de persistance. Ainsi aucune requête SQL ne doit être réalisée dans la couche « entreprise ».

Deux solutions sont alors envisageables, créer sa propre couche de mappage relationnel/objet, ou utiliser un outil de mappage existant.

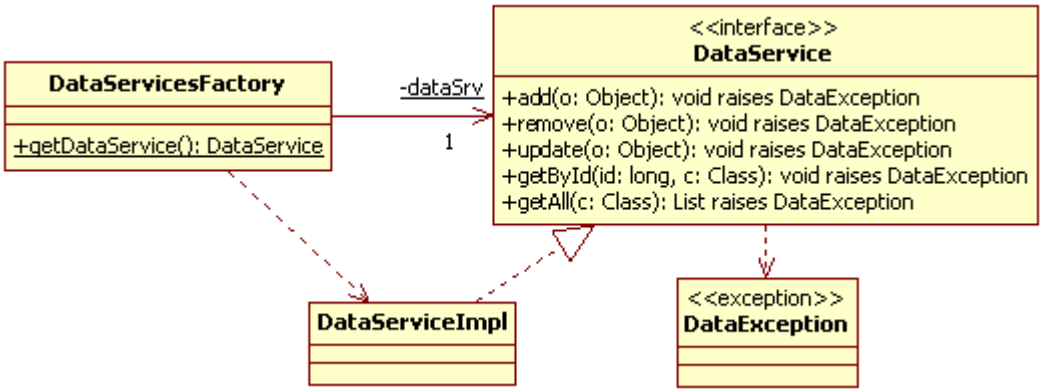
La création de la couche de mappage relationnel/objet peut être plus ou moins complexe en fonction de la nature des objets à rendre persistants, et des relations qui les lient (associations entre classes). A l'exception de cas simple, le recours à un outil de mappage s'avère nécessaire.

Nous optons pour notre application, d'utiliser les dernières spécifications java en la matière, et plus précisément JPA (Java Persistance API), que nous décrirons prochainement, et qui comme nous le verrons, permettra de faciliter énormément le codage grâce en particulier à l'utilisation des annotations.

Par manque de place, nous ne traitons pas des problèmes d'optimisation de l'accès aux données par l'emploi de critères de recherche particuliers. Les méthodes offertes par notre interface « DataService » sont donc réduites au minimum.

On remarquera l'indépendance vis à vis de la couche supérieure, les règles du 3-tier sont bien respectées. Maintenir cette indépendance complique en général le processus de modélisation. Le problème est que la couche « entreprise » a besoin de passer à la couche « data » les objets métiers à rendre persistants. Or la séparation des couches impose que la couche inférieure ignore tout de la couche supérieure. Ceci oblige donc à une modélisation plus complexe.

Une autre solution qui permet de simplifier le problème est de considérer que les classes entités sont figées dans leur structure. En cas d'évolution de l'application de gestion de la médiathèque, ce sont les services métiers qui sont le plus susceptibles d'évoluer et non, à priori, les objets métiers. Les classes entités peuvent être donc perçues comme des classes d'une bibliothèque à laquelle toutes les couches peuvent accéder. Il faut cependant être conscient, que cette simplification dans la conception, peut impacter la séparation des couches, en introduisant une dépendance entre la couche données et une partie de la couche entreprise (les entités). Toute modification dans les classes entités impacte donc les deux couches, ce n'est pas ce qui est souhaité. Ceci montre, une fois de plus, l'importance primordiale de la bonne modélisation des classes entités.



Conclusion provisoire

La conception détaillée de notre application, réalisée en se basant sur le paradigme 3-tier, comprenant les couches client, entreprise et data, nous a permis à ce stade de définir les couches « entreprise » et « data » et les services qu'elles rendent. Notre prochaine étape sera d'élaborer la couche « client ». Pour cela l'utilisation d'un cycle de prototypage sera mieux adapté. Notre client sera une application Web, mais tout le socle pour le développement d'un client classique, de type

« desktop » est présent. Notre prochaine étude portera sur le développement d'une application Web avec NetBeans 6.0.

Troisième partie - JPA et application Web

Nous arrivons à la dernière partie de la réalisation de la gestion d'une médiathèque, dont le sujet de base est l'analyse et la conception objet d'une application.

Nous avons montré comment, en s'appuyant sur des règles de conception et notamment sur le modèle 3-tier, modéliser les classes et nous sommes ainsi parvenu à la définition de deux des 3 couches du modèle, les couches métier et données.

Cet article se focalise sur deux points, l'utilisation de JPA (Java Persistance API) pour assurer la persistance des données et sur la couche cliente.

Bien entendu, nous recommandons à nos lecteurs les précédents articles qui nous ont permis de spécifier toute notre application en utilisant la modélisation UML. Pour mémoire, notre application de gestion d'une médiathèque doit permettre à des adhérents d'emprunter des médias, après s'être acquitté du paiement d'une cotisation.

La persistance des données avec JPA

Le JDK 5 introduit une API pour la persistance des objets, appelée JPA (Java Persistance API). Son intérêt est qu'elle utilise de simples POJO (Plain Old Java Object). Inutile donc de dériver les classes entités de classe mère, ni de définir un fichier XML pour réaliser le mappage relationnel/objet. La nouvelle API utilise les annotations qui sont une nouveauté introduite dans le JDK 5. Le deuxième intérêt est que cette API fonctionne à la fois avec les éditions EE et SE de java.

De simples POJO ? Pas tout à fait, car en plus des annotations, il faut importer les classes correspondantes. Mais avec l'aide de l'IDE NetBeans, tout ceci est un jeu d'enfant. Ajouter une annotation et l'IDE sous signale un problème et vous propose d'importer la bonne classe.

JPA impose de plus les contraintes supplémentaires suivantes:

- la classe doit être sérialisable
- la classe doit avoir un constructeur sans argument avec une visibilité **public** ou **protected**
- la classe doit avoir un attribut permettant d'identifier de façon unique chaque instance, et servant de clé primaire

Voici à titre d'exemple le code source de la classe entité « **Emprunt** »

```
package entreprise;

import java.io.Serializable;
import java.util.Calendar;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Temporal;

@Entity
public class Emprunt implements Serializable {

    @Id
    @GeneratedValue
    private long id;

    @OneToOne
    Exemplaire exemplaire;

    @OneToOne
    Adherent adherent;

    @Temporal(javax.persistence.TemporalType.DATE)
    Date dateEmprunt;

    protected Emprunt() {
    }

    Emprunt(Exemplaire e, Adherent a) {
      if (e == null || a == null) {
```

```
            throw new IllegalArgumentException();
        }
        this.exemplaire = e;
        this.adherent = a;
        Calendar cal = Calendar.getInstance();
        cal.set(cal.get(Calendar.YEAR),cal.get(Calendar.MONTH),cal.get(Calendar.DAY_OF_MONTH),0,0,0);
        cal.set(Calendar.MILLISECOND, 0);
        this.dateEmprunt = cal.getTime(); // la date à 00:00:00
    }

    public Adherent getAdherent() {
        return adherent;
    }

    public Date getDateEmprunt() {
        return dateEmprunt;
    }

    public Exemplaire getExemplaire() {
        return this.exemplaire;
    }

    public Media getMedia() {
        return this.exemplaire.media;
    }

    public long getId() {
        return this.id;
    }

    public long getNumeroExemplaire() {
        return this.exemplaire.id;
    }
}
```

La signification des annotations est la suivante:

- @Entity indique que la classe est persistante
- @Id indique que l'attribut « id » est utilisé comme clé primaire
- @GeneratedValue indique que la valeur de « id » sera automatiquement générée
- @OneToOne indique la multiplicité de l'association (relation 1-1)
- @Temporal précise la correspondance entre l'objet Date et le type SQL utilisé

Nous remarquons de plus que la classe est bien sérialisable, et qu'elle possède un constructeur sans argument, nécessaire à JPA pour créer l'instance de l'objet depuis la source de données.

Les règles de conception que l'on s'impose, pour les classes entités, dans le modèle 3-tier, sont les suivantes:

- on délègue à la classe de service qui gère l'entité (par exemple *EmpruntService* pour la classe entité Emprunt), la responsabilité de la création des instances de classes. Ceci permet d'appliquer si besoin des règles métiers dans le processus même de création. Pour cela il n'y a pas de constructeur public. Le service métier utilise le constructeur de type package, puisqu'il est situé dans le même package. Le constructeur sans argument nous est imposé par JPA, et nous lui donnons la visibilité protected, la seule autorisé en dehors de la visibilité public, que l'on veut éviter.
- tous les attributs de la classe entité ne sont accessibles qu'en lecture, en dehors du package « entreprise », par contre et sauf exception (comme pour l'id qui est une valeur autogénérée), ils sont tous accessibles en écriture pour la classe de service. La modification de l'objet est à nouveau déléguée à la classe de service, ce qui permet d'en contrôler l'autorisation et la validité, ansi que la répercusion de cette modification sur la cohérence de l'ensemble des données
- les attributs modifiables par la couche client, sont aussi délégués à la classe de service, ceci revient à déporter les « setters » dans la classe de service

Les méthodes de la classe de service:

- Entity newEntity() crée un objet entité temporaire
- void add(Entity e) rend l'objet persistant

- void update(Entity e) modifie un objet persistant
- void remove(Entity e) supprime un objet persistant
- void setAttribute(Entity e, <Type> value) modifie temporairement un attribut

Le framework JSF Visual Web de Netbeans

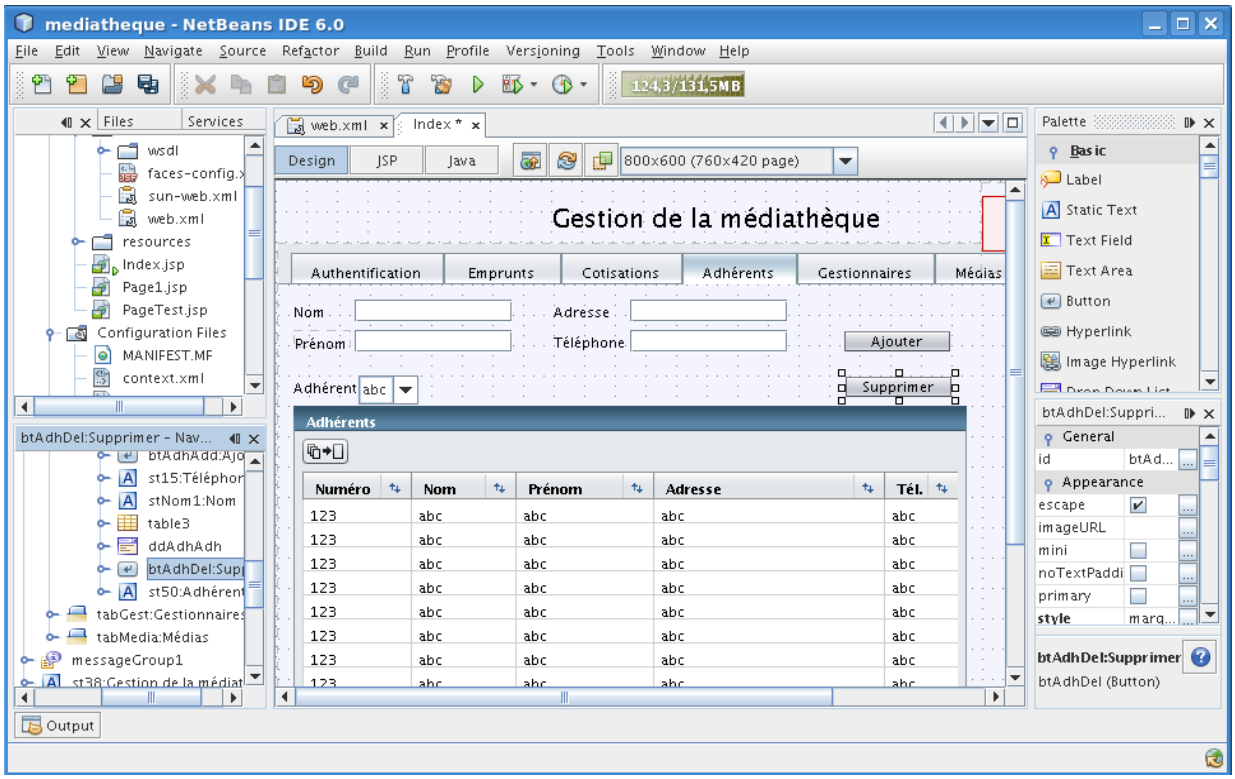
La couche cliente de notre réalisation est une application web, développée avec NetBeans 6.0 et plus précisément avec son module « Visual Web ». Selon la version de NetBeans utilisé, ce module est présent ou devra être installé en tant que module supplémentaire (plugin). NetBeans permet directement à partir de son environnement de développement, de télécharger des modules ou de faire des mises à jour.

Le concepteur RAD de page web

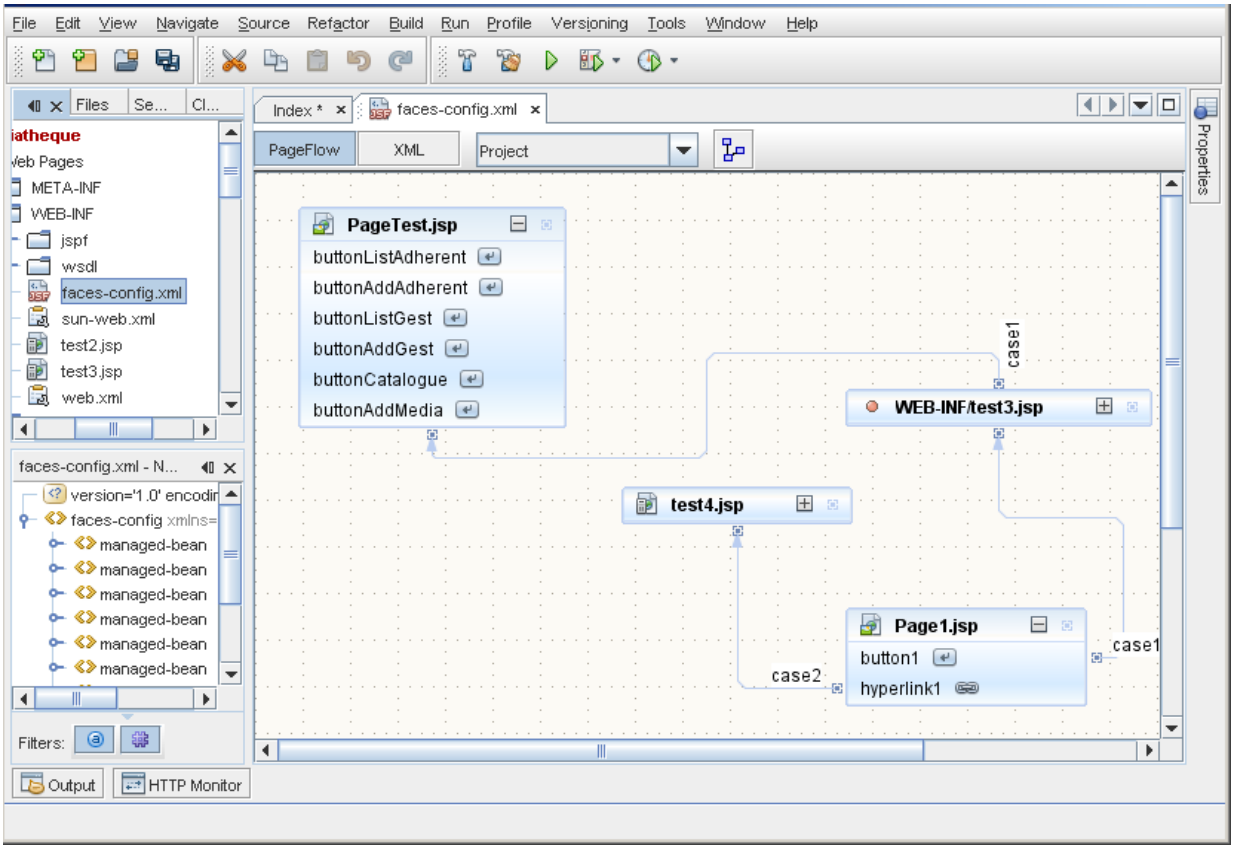
« Visual Web » est un concepteur RAD sophistiqué de pages web. Il offre une palette de composants permettant de construire aisément et rapidement l'interface graphique de l'application web, ainsi qu'un outil graphique permettant de définir la navigation entre les pages.

On peut dire que « Visual Web » permet en fait de faire un mappage IHM / Objet de manière quasi transparente.

L'environnement de développement

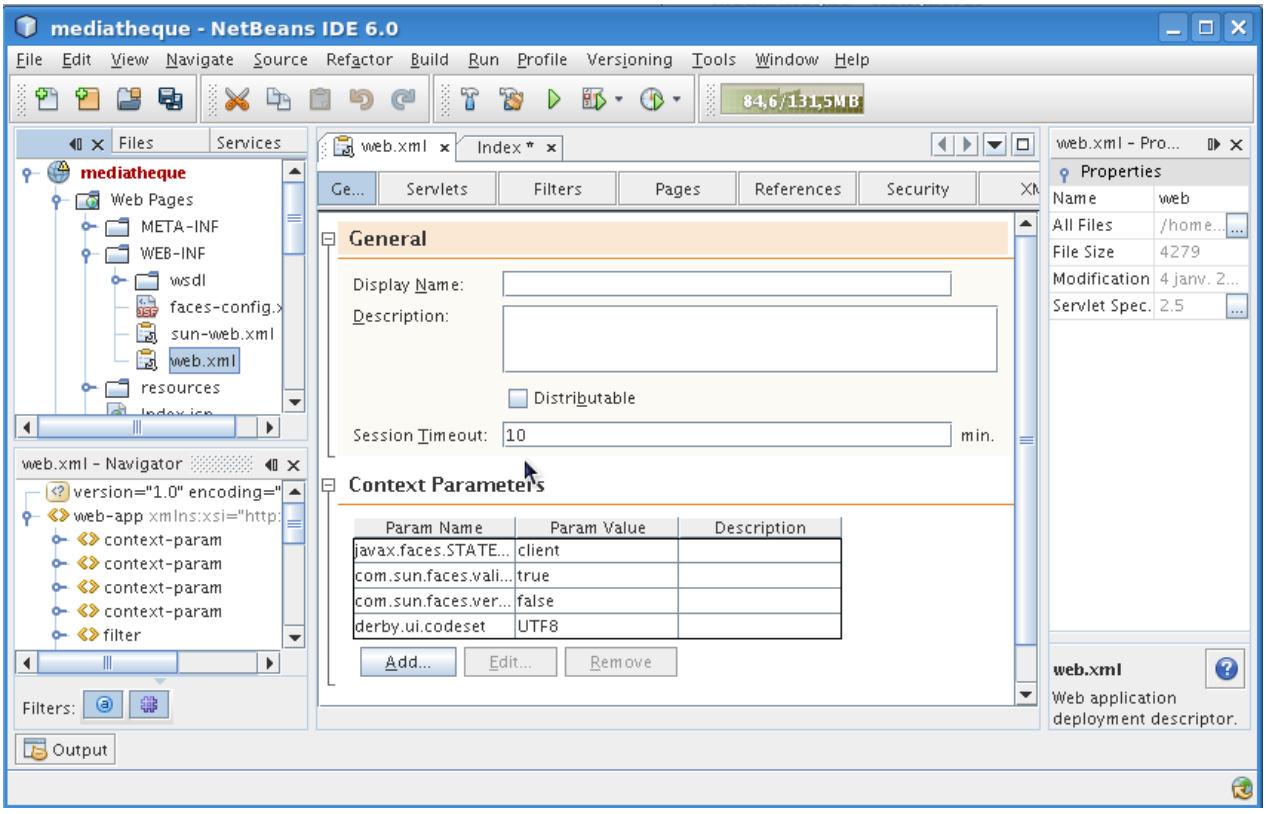


Le concepteur de navigation entre les pages Web



Le concepteur de navigation permet de définir de manière graphique les règles de navigation entre les différentes pages de l'application Web. Toutes les informations correspondantes sont stockées au format XML, dans le ficher « **faces-config.xml** ». Ce dernier peut aussi être édité manuellement.

Le fichier de déploiement « web.xml »



Le fichier de déploiement classique « web.xml » est visualisé en mode graphique, ce qui est très pratique. Bien entendu l'édition directe du fichier xml est toujours possible. Quelques paramètres importants peuvent être définis à ce niveau. Le champ « session timeout », indique le temps au bout

duquel une session sera fermée en cas d'inactivité, la valeur par défaut est de 10 minutes. Cette donnée est importante, pour les applications qui nécessite une connexion avec authentification. Un autre champ important est « Context parameters », il permet au serveur d'application (conteneur de servlets) de communiquer des paramètres initiaux aux servlets, que celles-ci peuvent récupérer à l'aide de la méthode getInitParameter(). L'utilisation de la base javaDB (apache Derby) permet de spécifier l'encodage des données par l'intermédiaire du paramètre « derby.ui.codeset ». Ce paramètre est initialisé à « UTF8 » pour utiliser l'encodage UTF-8. Consulter le lien http://db.apache.org/derby/docs/dev/tools/rtoolsijpropref97949.html pour la description détaillée de ce paramètre.

Le serveur d'application (GlassFish, Tomcat ou autre)

Netbeans 6.0 propose par défaut le serveur d'application GlassFish V2. Pour rester plus classique notre application utilise Tomcat, et nous détaillerons la configuration particulière à utiliser avec ce serveur. Cependant elle fonctionne sans difficulté avec GlassFish. Le choix du serveur est bien entendu configurable grâce à la feuille de propriétés du projet.

Gestion de la persistance des classes entités avec NetBeans

Notre exemple se base sur l'utilisation de la base Apache Derby (JavaDB) en mode embarquée, mais il est parfaitement possible de l'utiliser en mode autonome, ou même d'utiliser toute autre base dont le driver JDBC est disponible, avec le support JPA.

L'accès aux données est définie grâce au fichier **persistance.xml**. La notion importante est celle d'unité de persistance: «**Persistence Unit** ». L'unité de persistance est le nom logique qui sera utilisée pour accéder à une source de données particulière. L'unité est associé et fait référence au framework de mappage relationnel/objet qui sera utilisé. Par défaut, c'est le framework TopLink qui est proposé, mais il est tout à fait possible, d'installer et d'utiliser un autre framework tel que Hibernate. Le framework en tant que tel est appelé « **Service provider** ». Enfin il faut indiquer la source de données elle-même et le type de base de données utilisé (Derby, MySql ...), désigné par « **Data source** ». Selon que l'on utilise JTA ou non, « Data source » représente le nom JNDI de la source de données ou directement le driver JDBC. Dans ce cas on ne bénéficie pas de la gestion d'un pool de connexions comme peut l'offrir un serveur J2EE.

Pour schématiser: **Persistence Unit -> Service provider -> Data source**

Exemple de « Persistence Unit » dans le cas d'utilisation d'un serveur J2EE (Glassfish)

```
<persistence-unit name="mediathequePU" transaction-type="JTA">
  <provider>oracle.toplink.essentials.PersistenceProvider</provider>
  <jta-data-source>mediatheque</jta-data-source>
  <properties>
    <property name="toplink.ddl-generation" value="create-tables"/>
  </properties>
</persistence-unit>
```

Le choix de Tomcat ne nous permet pas d'utiliser JTA. Il faut dans ce cas, éditer manuellement le fichier persistance.xml. Voici la définition utilisée pour l'application de gestion de la médiathèque.

```
<persistence-unit name="mediathequePU" transaction-type="RESOURCE_LOCAL">
  <provider>oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider</provider>
  <jta-data-source/>
  <class>entreprise.Adherent</class>
  <class>entreprise.Gestionnaire</class>
  <class>entreprise.Media</class>
  <class>entreprise.Emprunt</class>
  <class>entreprise.Cotisation</class>
  <class>entreprise.PaiementCotisation</class>
  <class>entreprise.Exemplaire</class>
  <exclude-unlisted-classes>true</exclude-unlisted-classes>
  <properties>
   <property name="toplink.jdbc.url" value="jdbc:derby:databases/mediatheque;create=true"/>
    <property name="toplink.jdbc.user" value=""/>
    <property name="toplink.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
    <property name="toplink.jdbc.password" value=""/>
    <property name="toplink.ddl-generation" value="create-tables"/>
  </properties>
</persistence-unit>
```

On remarque notamment que le type de transaction est différent, JTA ou RESOURCE_LOCAL, ainsi que le provider qui n'est pas le même. Il faut de plus dans notre cas, préciser explicitement toutes les classes persistantes. Nous utilisons la base Derby embarqué, ceci est spécifié par le nom du driver JDBC « org.apache,derby.jdbc.EmbeddedDriver ». L'option « create=true » précise que les tables seront automatiquement créés au démarrage du serveur, les fichiers correspondants seront placés dans le répertoire « databases/mediatheque », relativement au répertoire courant du serveur Tomcat. L'option « create=true » pourra être retirée après le premier déploiement, mais ceci n'est pas critique dans la mesure où l'erreur qui sera généré, en cas d'existance des tables, peut être ignorée, sachant de plus, que le redémarrage du serveur ne se produit que de façon exceptionnelle, en production.

La couche « client »

La couche « client » comprend toutes les applications clientes. Dans le cas de la médiathèque, il existe deux applications, l'une destinée à l'adhérent pour la consultation du catalogue et la réservation en ligne des médias, l'autre pour les gestionnaires chargés de l'administration du système et des entrées/sorties des médias. Le sujet est donc vaste, et nous ne pourrons pas le traiter dans son intégralité dans cet article. Nous nous focaliserons uniquement sur l'application d'administration destinée aux gestionnaires.

Après une phase d'authentification, le gestionnaire peut effectuer les opérations suivantes:

- gérer les adhérents (ajout, suppression ...), ainsi que les gestionnaires
- gérer les médias et les exemplaires
- gérer les emprunts selon divers critères (emprunts par adhérent, emprunts par médias ...)

A la première utilisation, la base étant vide, et aucun gestionnaire n'étant défini, la phase d'authentification est problématique. Nous optons, dans ce cas, pour la solution qui consiste à utiliser des valeurs par défaut lorsque la liste des gestionnaires est vide dans la base, par exemple « admin » pour le login et le mot de passe.

L'application Web et les pages JSF

Lors de la création d'une application Web, avec « Visual Web » dans NetBeans, l'IDE génère automatiquement trois classes « Bean » dont les instances ont des cycles de vie différents. Ces beans sont appelés « managed beans »,car ils sont gérés directement par le framework, l'application ne doit pas les instancier. Ces « managed beans » sont les suivants:

- « ApplicationBean1 » : une seule instance de cette classe est créée au premier accès à l'application et persiste durant tout le temps où l'application reste déployée sur le serveur (le cycle de vie est de type application)
- « SessionBean1 » : une nouvelle instance de cette classe est créée à l'ouverture d'une session, et persiste uniquement le temps de la session, l'objet est détruit à la fin de la session (le cycle de vie est de type session)
- « RequestBean1 » : une nouvelle instance de cette classe est créée suite à une requête HTTP, et persiste uniquement le temps de la requête (le cycle de vie est de type requête)

Le développement d'une application web consiste ensuite à ajouter des pages JSF, puis à définir les règles de navigation entre ces pages. A chaque ajout d'une nouvelle page, l'IDE crée deux fichiers, un fichier jsp et une classe java Bean qui hérite de AbstractPageBean. Une nouvelle instance de cette classe est crée à chaque invocation de la page jsp, le cycle de vie est de type page, et il est le plus court. Ne pas confondre avec le cycle de vie de type requête, car pour ce dernier plusieurs pages peuvent être invoquées si la première fait une redirection vers une autre page, etc.

Les beans 'page', 'request' et 'session' disposent de la méthode « getApplicationBean1() » pour récupérer l'instance du bean 'application'. De même les beans 'page' et 'request' dispose de la méthode « getSessionBean1() pour obtenir l'instance du bean 'session'.

Selon la portée que l'on voudra donner à un objet, on le stockera en tant qu'attribut de l'une des quatre classes précédentes.

Pour construire la page, le développeur dispose d'un concepteur graphique qui permet de déposer les composants graphiques issu de la palette d'outils, le fichier jsp est mis à jour automatiquement et peut être ignoré. Après construction de l'interface graphique, le développement consiste essentiellement à écrire le code java correspondant à la gestion événementielle.

Pour notre application, nous utilisons une seule page (fichiers Index.java et Index.jsp), et nous optons pour une présentation sous forme d'onglets.

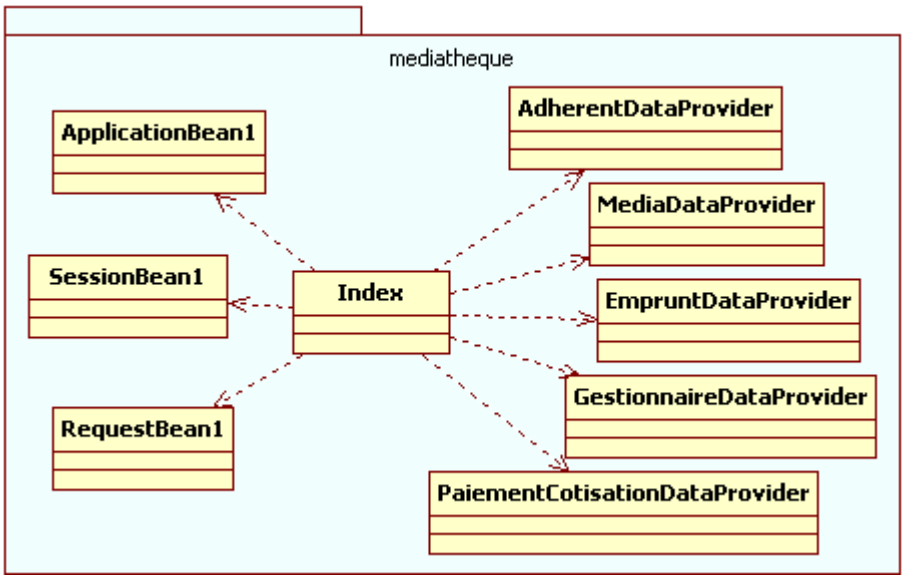
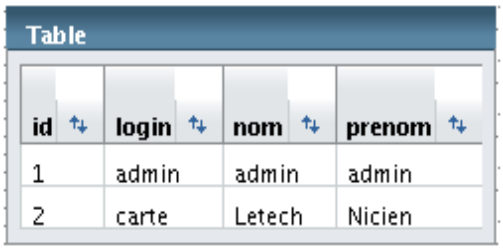


Diagramme de classe de la couche client

Le diagramme de classes du sous paquetage client « mediatheque »

Les données stocker dans la base de donnée, comme les adhérents par exemples, seront affichées dans la page Web grace à des objets Table.



Le composant JSF Table

En effet, le composant JSF « Table », permet de représenter des données dans une table.

Sont remplissage est « semi-automatique ».

Pour cela, le composant JSF « Table » doit être lié (binding) à un objet « DataProvider », qui est sensé lui fournir les données à présenter.

Les « DataProvider » que l'on utilise sont des classes dérivés de ObjectArrayDataProvider.

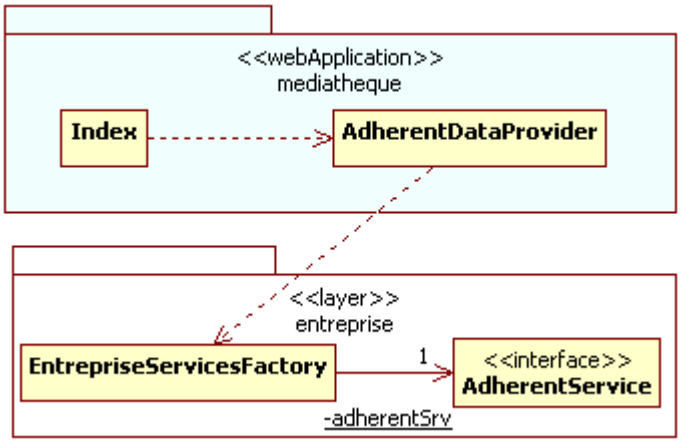
De plus, certaines règles sont à respecter afin que le DataBinding puisse lier les données comprises dans des objets métiers de manières automatique dans les objets graphiques comme Table et DropDown par exemple.

En effet, les classes entité doivent respecter des règles de nommage ISO pour les méthode get et set (getNomAttribut et setNomAttribut).

Pour afficher un tableau d'objets dans un composant JSF « Table », procéder comme suit:

- créer une classe « DataProvider » qui dérive de ObjectArrayDataProvider dans la couche cliente (attention en JSF cette couche porte le nom de l'application !)
- utiliser la méthode setArray(Object[] array) de cette classe pour initialiser le tableau
- associer le « DataProvider » avec le composant JSF « Table », cette opération est nommé « binding »

- paramétrer le composant JSF, pour associer les colonnes de la table avec les attributs des objets dans le tableau
- le framework invoquera le moment venu la méthode « getArray() » pour obtenir le tableau
- Appeler les méthodes du DataProvider dans la couche cliente, qui elle appelle les méthodes de la couche inférieure, afin de mettre automatiquement à jour la table.



Les dépendances de classes.

Pour la classe Index le provider AdherentDataProvider est un « *décorateur* », au sens des design patterns, de la classe AdherentService.

Mais voyons cela de plus prés avec un exemple avec la classe entité Emprunt.

Notre classe doit avoir un nom évocateur formé comme cela : *NomObjetMetierDataProvider* soit ici EmpruntDataProvider dont le code se trouve ci-dessous.

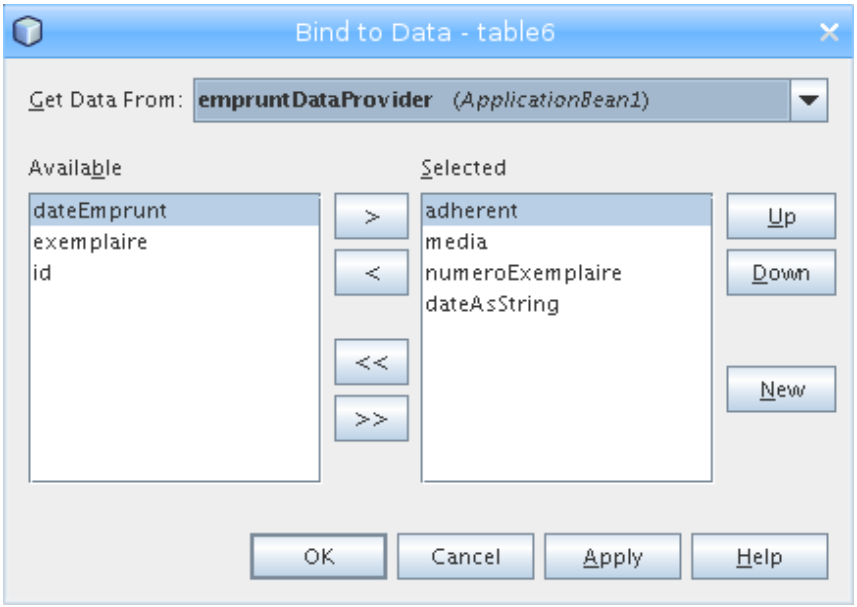
```
public class EmpruntDataProvider extends ObjectArrayDataProvider  {
  private Emprunt[] defaultObj = new Emprunt[0];

  public EmpruntDataProvider() {
    this.setArray(defaultObj);
  }

  public void setData(Object[] emprunts) {
    this.setArray(emprunts);
  }
  public void rendre(long expl) throws EntrepriseException {
    EmpruntService es = EntrepriseServicesFactory.getEmpruntService();
    Exemplaire ex = es.getExemplaireById(expl);
    if(ex != null)
      es.rendre(ex);
  }

  public Exemplaire getExemplaire(String id) {
    Exemplaire exp = null;
    if(id != null) {
      try {
        long id_ = Long.parseLong(id);
        EmpruntService ms = EntrepriseServicesFactory.getEmpruntService();
        exp = ms.getExemplaireById(id_);
      } catch(Exception e){}
    }
    return exp;
  }
}
```

Afin que de rendre notre DataProvider disponible dans le module RAD de binding, il nous faut créer un attribut EmpruntDataProvider dans un des Beans de la couche client.

Attention la portée du DataProvider varie en fonction du beans choisie ! Afin de le rendre disponible pour toute l'application, nous ajoutons nos DataProvider dans ApplicationBean1.java grâce au code suivant :

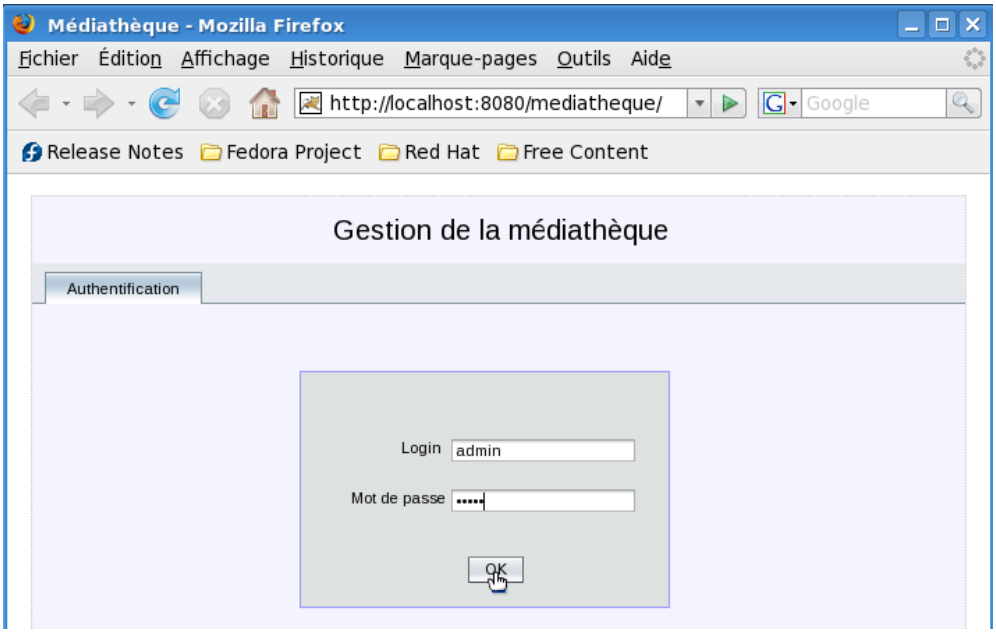
```
private EmpruntDataProvider empruntDataProvider = new EmpruntDataProvider();

public EmpruntDataProvider getEmpruntDataProvider() {
        return empruntDataProvider;
}
```

Il ne nous reste plus qu'a faire le Binding, pour cela un click droit en mode design sur l'objet table, puis sur l'option « Bind To Data » nous affiche le module de binding graphique ci-dessous.



Remarques :

1. on peut choisir les attributs qui doivent apparaître dans la table. S'il sont automatiquement présent, cela est due au fait que la classe entité respecte les conventions de nommage (get et set pour l'accès aux attributs), d'où l'interet de leurs respects.
2. Le fait d'ajouter à notre DataProvider, « l'ensemble des méthodes » de service de l'interface EmpruntService (du moins celle susceptible de modifier l'affichage), permet lors d'un appel via le DataProvider dans le code java de la page de mettre à jour automatiquement le contenue affiché par l'objet Table.



La page de connexion à l'application

La présentation sous forme d'onglets

Le suivi de session

Le suivi de session est une opération indispensable afin de garantir l'accès sécurisé à l'application d'administration. L'application a, en effet, besoin de savoir si une session est toujours active ou non, et de réclamer une nouvelle authentification si nécessaire. Voici un exemple de code permettant de réaliser cette fonction:

```
public void init() {
  // ...

  ExternalContext ctx = this.getExternalContext();
  HttpSession httpSession = (HttpSession) ctx.getSession(true);
  String id = httpSession.getId();
  SessionBean1 sessionBean = this.getSessionBean1();
  String oldId = sessionBean.getSessionId();
  sessionBean.setSessionId(id);
  boolean logged = id.equals(oldId) && sessionBean.isLogged();

  // ...
}
```

Le principe est le suivant:

- utiliser la méthode « getExternalContext() » pour obtenir le contexte de la requête courante
- utiliser la méthode « getSession() » du contexte pour obtenir l'objet session, l'argument 'true' crée la session si elle n'existe pas encore, sinon la méthode retourne 'null'
- utiliser la méthode « getId() » de l'objet session pour obtenir l'identifiant unique de la session
- sauvegarder l'identifiant dans le bean session (SessionBean1)

On ajoute dans le bean session, deux propriétés (attributs): l'identifiant de la session, et une information qui indique si un gestionnaire a été authentifié avec succès, dans notre cas, il s'agit d'un booléen. Voici le code ajouté dans la classe SessionBean1:

```
  private boolean logged;
  public boolean isLogged() {
    return this.logged;
  }
  public void setLogged(boolean logged) {
    this.logged = logged;
  }

  private String sessionId;
  public void setSessionId(String sessionId){
      this.sessionId = sessionId;
  }
  public String getSessionId(){
      return this.sessionId;
  }
```

En cas de perte de session, une nouvelle instance du bean session est crée, et les attributs « logged » et « sessionId » sont initialisés à leur valeur par défaut, 'false' pour le premier et 'null' pour le second. Ainsi l'application peut aisément contrôler la perte de connexion.

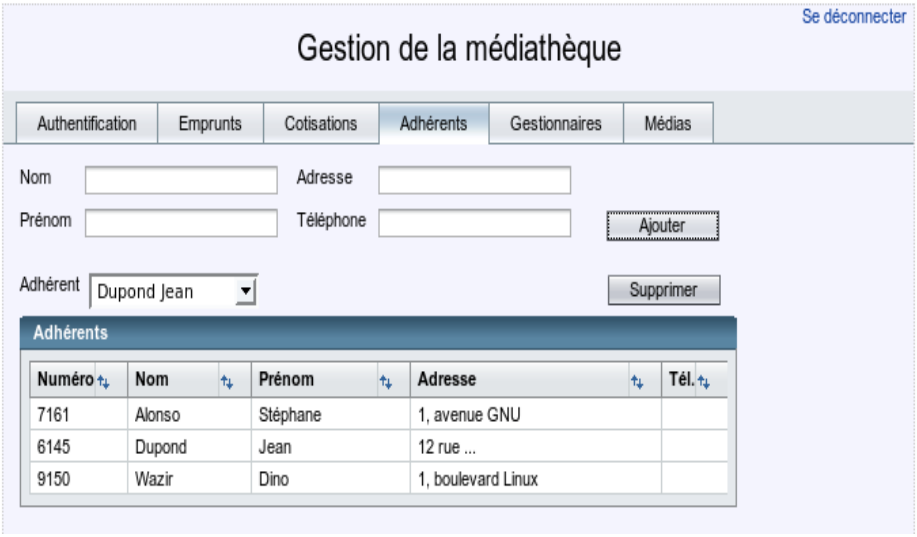
La propriété « logged » peut être avantageusement remplacé par le login, car l'application peut avoir besoin de connaître quel est la personne connecté.

Réalisation et test

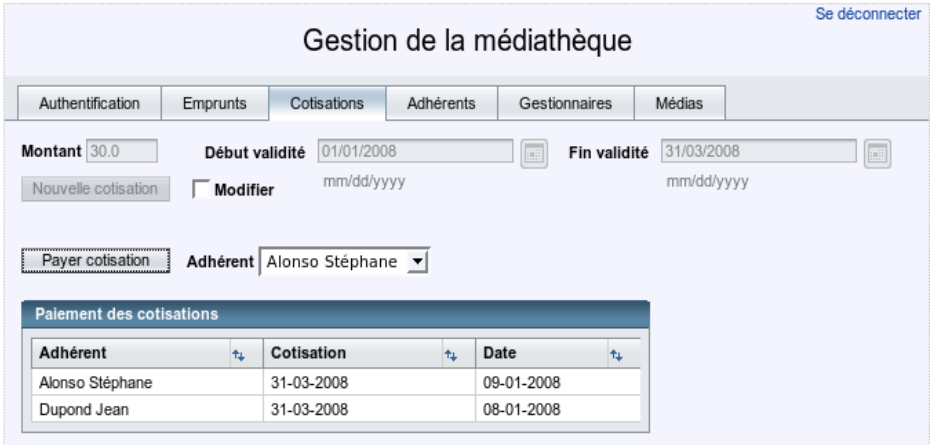
Tests unitaires

NetBeans est capable de générer automatiquement des classes de test pour effectuer des tests unitaires, en se basant sur JUnit. Il ne nous est pas possible de détailler cette question dans le cadre de cet article, mais nous invitons le lecteur à explorer cette fonctionnalité.

L'application finale



La gestion des adhérents



La gestion des cotisations

La gestion des médias

Conclusion

Nous arrivons au terme de notre étude, au cours de laquelle nous avons tenté d'élaborer une méthodologie et des règles de conception objet, permettant de favoriser la qualité du développement logiciel en équipe. Les points clés mis en avant sont la modularité et la recherche de dépendances minimale entre classes et paquetages, dans une optique de favoriser l'évolutivité du logiciel et l'indépendance maximale vis à vis de composants externes.

Liens utiles

http://www.netbeans.org/
http://db.apache.org/derby
http://java.sun.com/developer/technicalArticles/J2EE/jpa/

Contacter les auteurs

alonso.stephane27@free.fr
dwazir@free.fr

Sites Web

http://alonso.stephane27.free.fr

Ou faire un site web commun avec l'article et l'application à télécharger ?