Cabmed

Gestion d'un cabinet médical

Présentation générale du mémoire :

Cabmed est un projet composé de trois applications différentes qui "interagissent" entre-elles. Cette application (ou ensemble d'application) sert à la gestion d'un cabinet médical. **J'insiste sur le fait qu'il s'agit-là de la gestion d'un CABINET MÉDICAL, pas d'un HÔPITAL !** Ici le projet est divisé en trois parties que j'ai appelé : "*Admin*" (gestion du personnel du cabinet), "*Manage*" (gestion des rendez-vous et dossier des patients) et enfin "*Web*" (ne servant qu'à consulter les rendez-vous passés et futurs). Ces trois parties sont trois applications différentes mais qui tournent autour d'une même base de données bien évidemment, le but étant de les faire fonctionner ensemble. La base de données tourne sur un serveur *MySQL*.

Avant de continuer je précise que j'ai volontairement choisi de ne pas m'occuper de certains points comme la comptabilité et la gestion des stocks (fournitures, médicaments, ...), ces points-là étant moins orientés informatique...

Chemin emprunté et technologies choisis :

Pour cette application j'ai choisi de développer avec le langage de programmation Java. Étant un langage dont j'ai une certaine "maîtrise" (celui que j'utilise le plus aisément en tout cas) et proposant tout le nécessaire pour faire en même temps des applications orientés PC (applications *Admin* et *Manage*), mais aussi orienté web (comme l'application *Web* comme vous l'aurez compris).

Après le codage des interfaces graphiques (qui n'était pas si difficile que ça mais qui était assez gros à faire), le plus gros dans ce projet était la gestion du planning des médecins. Gérer un planning est l'une des choses les plus difficiles à faire. Ce qui rend cette tâche compliqué (à mes yeux en tout cas), c'est surtout l'enregistrement en base de données de ces informations, comment organiser la base de données de façon à pouvoir stocker tout le nécessaire. Nous avons certains

outils qui nous permettent de ne pas nous occuper de la base de données, l'outil la crée et la gère pour nous en se basant sur la structure de notre application (j'en parle au prochain paragraphe), mais en utilisant ces outils, on est obligé dans certains cas d'adapter notre application pour être compatible avec la base de données, ce qui nous aide pas tellement... J'ai donc opté pour *Google Agenda* ! Google propose un service d'agenda qui est assez bien fait et qui peut être adapté (jusqu'à un certain point bien sûr) selon nos besoins. Et Google fait encore mieux en nous proposant une librairie Java toute faite avec tout le nécessaire pour gérer cet agenda. Malgré qu'il existait déjà cette librairie, comprendre son fonctionnement et réussir à la prendre en main était assez compliqué (c'est la première fois que je l'utilise).

Le deuxième outil utilisé est *JPA*. Cet outil est ce qu'on appelle un *ORM* (*Object Relational Mapping*). Un *ORM* est un outil qui s'occupe pour nous de la base de données (l'outil dont je parlais dans le paragraphe précédent). *JPA* est l'*ORM* de base compris dans les librairies Java, il n'y a donc aucune librairie supplémentaire à rajouter pour l'utiliser. En vérité, lorsqu'on regarde en détail comment *JPA* fonctionne, on comprend qu'en réalité ce n'est pas réellement un *ORM*, c'est plutôt un outil qui permet d'utiliser un *ORM*... *JPA* est donc un ensemble de classe qui fait appel aux véritables *ORM*s pour nous. Il existe plusieurs "vrais" *ORM*s, les plus connus étant *Hibernate*, *TopLink*, *EclipseLink*, etc et eux gèrent réellement une base de données, mais *JPA* lui est à mettre au-dessus de ceux-là et appelle l'un de ces *ORM*s à notre place. L'avantage d'utiliser *JPA* plutôt qu'un *ORM* directement est double : premièrement, *JPA* est plus facile à utiliser et à comprendre que ces autres *ORM*s, mais surtout, grâce à *JPA*, on peut passer d'un *ORM* à l'autre en un instant. En effet, nous avons la possibilité de préciser à *JPA* quel *ORM* il doit utiliser via un fichier de configuration. Plus besoin d'apprendre à utiliser tous ces *ORM*s ! En apprenant à utiliser *JPA*, on peut utiliser tous les avantages de tous les *ORM*s (certains d'entre eux étant plus adaptés dans certains cas).

Le dernier framework utilisé est *Struts II*. Cet outil n'est utilisable que dans le cadre des applications web. En effet, il ne sert qu'à organiser la structure des pages web (dire qu'à partir de telle page on peut accéder à telle ou telle page, de faire une certaine action avant de transférer à une page, ...). *Struts II* propose d'autres petites options (comme l'internationalisation des pages web) dont on parlera plus tard. Plus de détails dans la partie application web.

Pour finir j'aimerais parler de l'environnement de développement (*IDE*) utilisé. Les deux *IDE*s qui sont sûrement les plus connus pour le développement Java sont *NetBeans* et *Eclipse*, *NetBeans* étant fait par ceux qui ont inventé le langage Java, et *Eclipse* est lui un projet communautaire et open-source. L'énorme avantage d'*Eclipse* est qu'il est à 200% adaptable, on peut tout modifier, rajouter autant de plug-ins que l'on veut, ... *NetBeans* l'est aussi mais pas au même point. J'ai malgré ça opté pour *NetBeans* ! Je préfère de loin *NetBeans* ne serait-ce que parce qu'il est plus clair qu'*Eclipse*. Sur *Eclipse*, je me perds souvent dans les options mais surtout, j'ai du mal à me retrouver dans le code. Sur *NetBeans*, tout n'est pas parfait mais je trouve les options déjà plus claires mais surtout, je ne perds plus dans mon code ce qui me fait gagner un temps fou ! L'un des autres *IDE*s assez connus pour le

développement Java est *IntelliJ IDEA* mais lui n'est pas gratuit (pour ne pas dire assez cher...). Il a cependant très bonne réputation et je n'en ai entendu que du bien.

Présentation général des applications :

Comme expliqué plus haut, le projet est coupé en trois applications :

L'application "Admin" est celle permettant de gérer le personnel interne du cabinet ("admin" pour "administration") et utilisable que par l'administrateur.

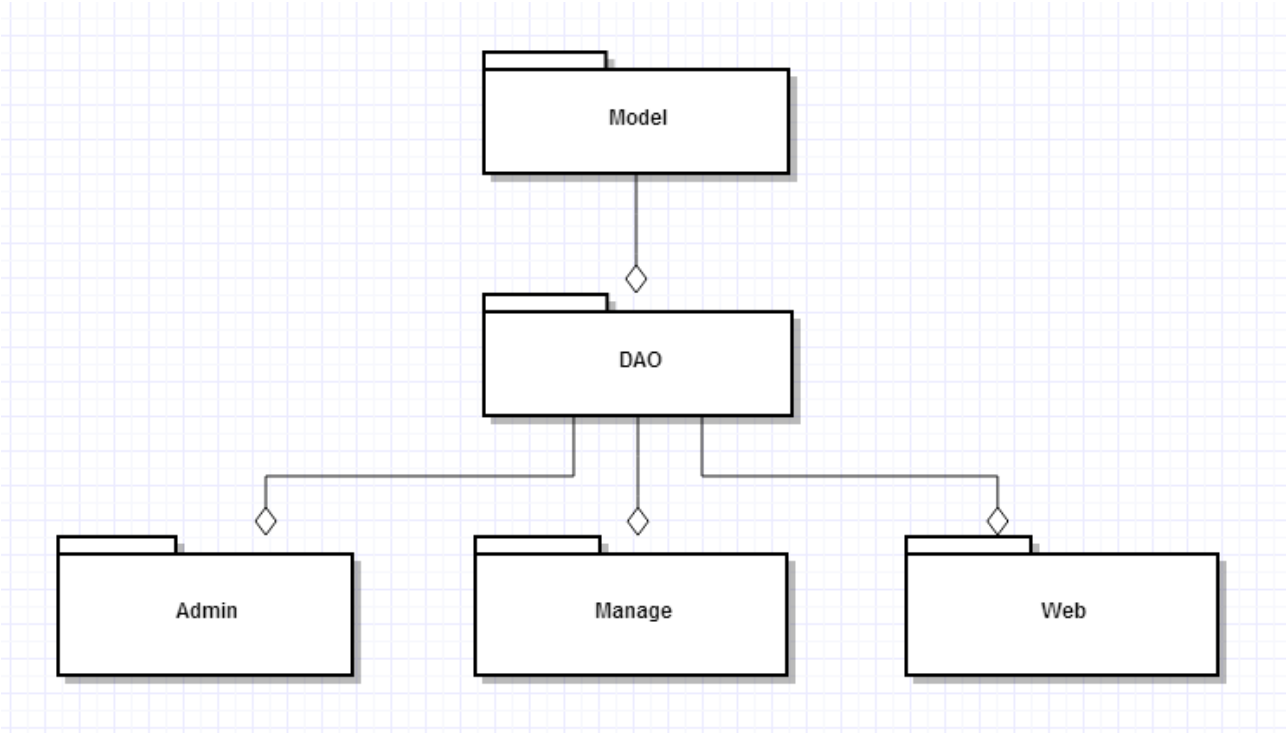
L'application "Manage" est celle permettant de gérer tout le reste et est utilisée par le personnel, les secrétaires et médecins.

L'application "Web" est celle permettant de consulter ses rendez-vous en ligne et utilisable par les patients et les médecins.

Structure des différents projets :

Structure DES projets :

Comme dit plus haut, le projet est divisé en trois applications, qui dans la pratique se divisent aussi en trois projets différents. En plus des trois projets pour les trois applications, j'ai rajouté deux autres projets, l'un étant un projet ne comprenant que ce que l'on appelle le "*modèle*" (tous les objets qui seront enregistrés dans la base de données), et l'autre contient ce que l'on appelle le "*DAO*" (pour ***D****ata* ***A****ccess* ***O****bject* => tous les objets servant à accéder et à gérer cette base de données). Voici un schéma qui résume la structure générale :

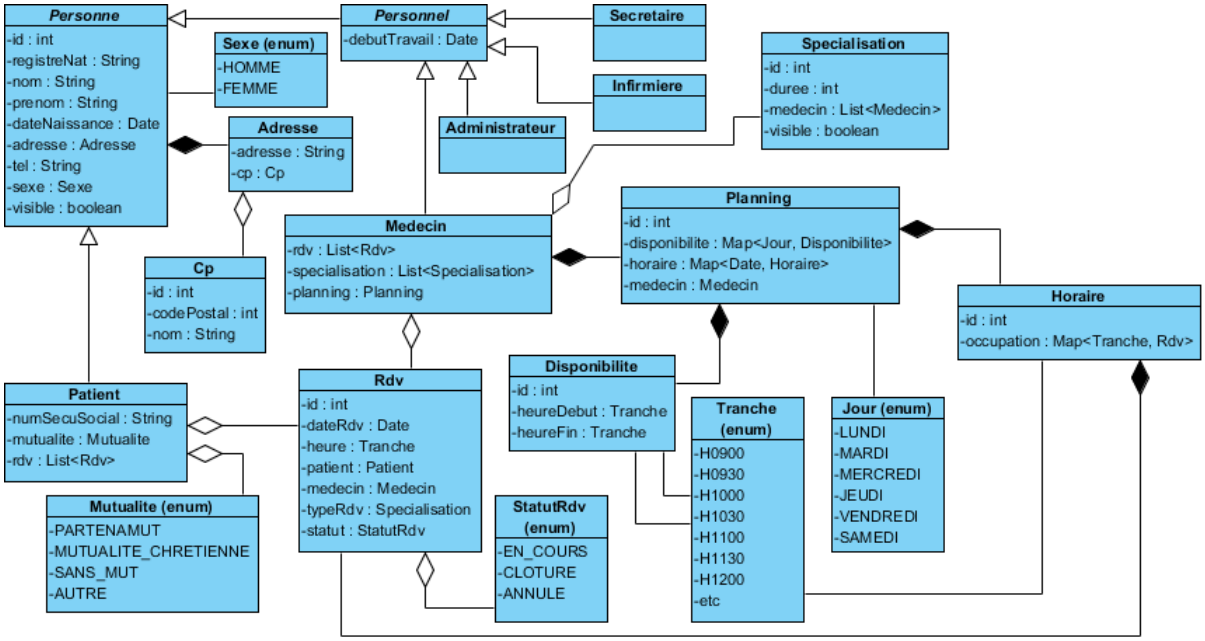


Ce qu'il faut comprendre ici c'est que chaque "dossier" (en Java on parle plutôt de "package") représente un projet spécifique. Le lien qui lie par exemple les projets *Model* et *DAO* signifie que le projet *Model* a été inséré dans le projet *DAO* (les objets servant à gérer la base de données ayant besoin de savoir ce qu'il faut y mettre...). L'insertion s'est faite une fois tout le modèle finie.

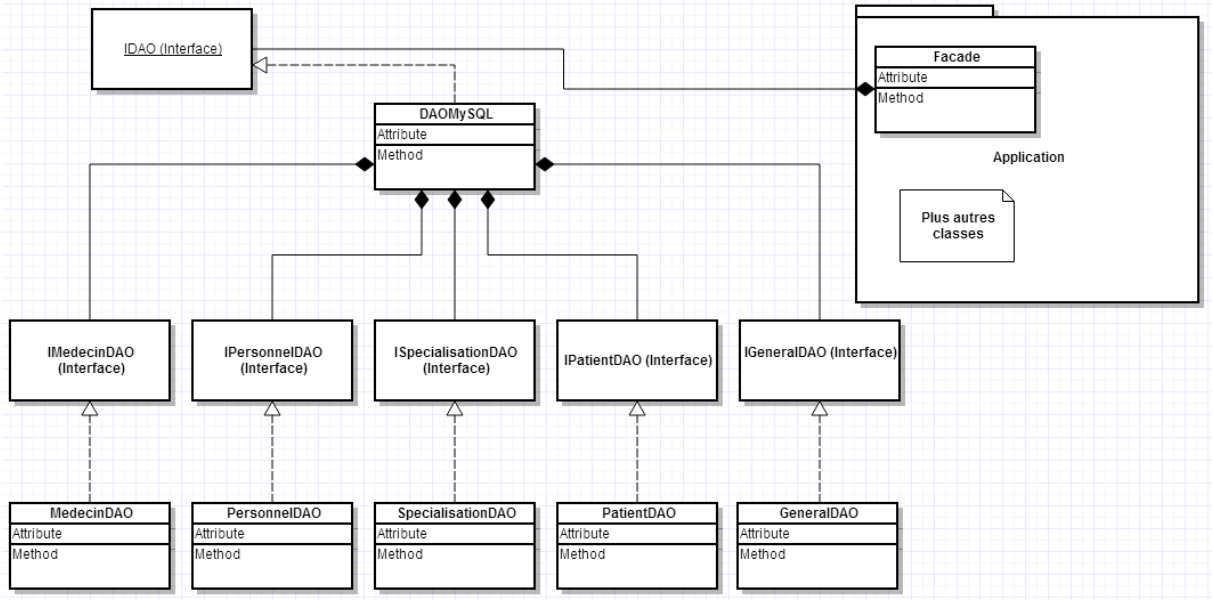
Si on comprend ça on peut donc comprendre que le projet *DAO* a été inséré dans les trois autres projets où je développais les applications. Étant donné que le projet *DAO* se trouve dans les trois projets d'applications, et que le projet *DAO* contient le projet *Model*, indirectement, les projets d'applications ont aussi accès au projet *Model*.

Voilà pour ce qui est de la structure des projets entre-eux. Ci-dessous je détaille chacun des projets pour présenter chaque classe qui intervient et à quoi sert chacune de ces classes...

Model :


Personne
-id : int
-registreNat : String
-nom : String
-prenom : String
-dateNaissance : Date
-adresse : Adresse
-tel : String
-sexe : Sexe
-visible : boolean
Personnel
-debutTravail : Date
Secretaire
Infirmiere
Administrateur
Sexe (enum)
-HOMME
-FEMME
Adresse
-adresse : String
-cp : Cp
Cp
-id : int
-codePostal : int
-nom : String
Specialisation
-id : int
-duree : int
-medecin : List<Medecin>
-visible : boolean
Medecin
-rdv : List<Rdv>
-specialisation : List<Specialisation>
-planning : Planning
Planning
-id : int
-disponibilite : Map<Jour, Disponibilite>
-horaire : Map<Date, Horaire>
-medecin : Medecin
Horaire
-id : int
-occupation : Map<Tranche, Rdv>
Patient
-numSecuSocial : String
-mutualite : Mutualite
-rdv : List<Rdv>
Mutualite (enum)
-PARTENAMUT
-MUTUALITE_CHRETIENNE
-SANS_MUT
-AUTRE
Rdv
-id : int
-dateRdv : Date
-heure : Tranche
-patient : Patient
-medecin : Medecin
-typeRdv : Specialisation
-statut : StatutRdv
StatutRdv (enum)
-EN_COURS
-CLOTURE
-ANNULE
Disponibilite
-id : int
-heureDebut : Tranche
-heureFin : Tranche
Tranche (enum)
-H0900
-H0930
-H1000
-H1030
-H1100
-H1130
-H1200
-etc
Jour (enum)
-LUNDI
-MARDI
-MERCREDI
-JEUDI
-VENDREDI
-SAMEDI

DAO :

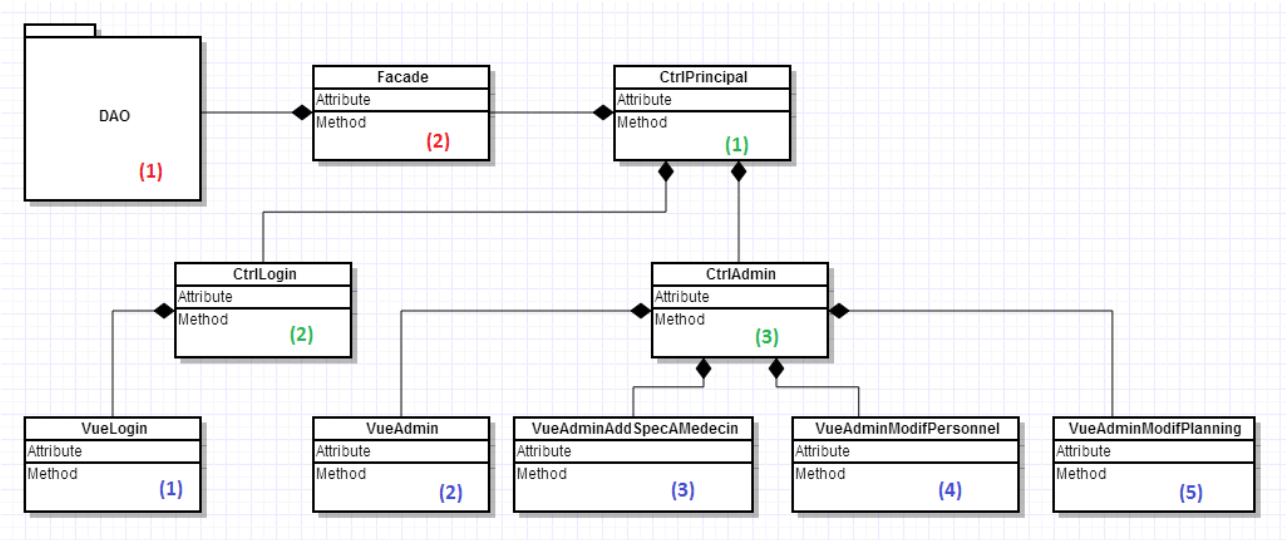

IDAO (Interface)
DAOMySQL
Attribute
Method
Facade
Attribute
Method
Application
Plus autres classes
IMedecinDAO (Interface)
IPersonnelDAO (Interface)
ISpecialisationDAO (Interface)
IPatientDAO (Interface)
IGeneralDAO (Interface)
MedecinDAO
PersonnelDAO
SpecialisationDAO
PatientDAO
GeneralDAO

Application Admin :

L'application *Admin* n'est peut-être pas la plus facile à développer (l'application web était légèrement plus facile) mais est sûrement la plus facile à expliquer (pour l'application web, il faut bien comprendre le fonctionnement du frameword *Struts II*). En effet elle fonctionne de manière assez simple. Ce qu'il faut savoir c'est que dans toutes applications Java, on retrouve une fonction qui est celle qui sera appelée lors du lancement de l'application et dans laquelle on prépare donc tout ce dont l'application a besoin pour être utilisée. On appelle cette fonction "*main*" (pas le choix au niveau de la structure de cette fonction, elle doit être écrite d'une manière bien spécifique sinon Java ne la reconnaît pas, par contre on fait ce que l'on veut dedans...). Dans cette application, ainsi que dans l'application *Manage* (les applications web fonctionnent différemment), cette fonction se trouve dans la classe "*CtrlPrincipal*". Il y a un gros souci avec une fonction *main()* c'est qu'elle est obligatoirement *static* (sans rentrer dans les détails du langage Java, disons qu'à cause de ça, cette fonction ne peut accéder à tous les champs de la classe dans laquelle elle se trouve... et comme dit plus haut, on ne peut changer ça sinon Java ne la reconnaît pas). Donc pour faire plus simple, j'ai utilisé la fonction *main()* juste pour générer une instance de *CtrlPrincipal* et j'ai fait tout le nécessaire dans son constructeur !

Dans cette application j'ai utilisé plusieurs *patterns* et *design pattern* (**Wikipedia** : « un **patron de conception** (en anglais : « design pattern ») est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels »). L'application respecte bien évidemment le schéma MVC, j'implémente le pattern Facade, Observer et Singleton dans certains cas, et comme dit plus haut, le projet DAO respecte le pattern DAO... Voir sous-chapitre sur les différents *patterns* utilisés.

Voici un schéma des différentes classes de l'application avec explications plus bas pour chacune de ces classes:



Général :

(1) – Cette image représente en fait le projet *DAO*, qui est un projet séparé du projet *Admin* mais qui a été intégré dedans pour pouvoir interagir avec la base de données.

(2) – La classe *Facade* se trouve dans le package "*cabmed.admin.main*". Elle est l'implémentation (comme son nom l'indique...) du pattern *Facade*. Dans mon cas, j'utilise une classe *Facade* pour faire un lien entre le *DAO* (accès à la base de données) et l'application elle-même. L'un des buts est de pouvoir implémenter par la suite le pattern *Observer* qui permet à toutes les vues de se mettre à jour à la moindre modification dans la base de données (pour cela il est important d'avoir une classe qui soit au courant de tout ce qui se passe avec la base de données, c'est la classe *Facade*...)

Contrôleurs : (package cabmed.admin.ctrl)

(1) – Comme déjà dit plus haut, la classe *CtrlPrincipal* est celle qui est appelée dès le lancement de l'application. C'est cette classe qui prépare tout pour que l'application puisse tourner. C'est dans cette classe que l'on recherche tout de suite la liste de tout ce qui sera utilisé dans l'application (liste de médecins, de secrétaires, d'infirmières et de spécialisations), on choisit le *look & feel* de l'application (le style d'affichage) et on instancie tous les autres contrôleurs pour qu'ils soient déjà disponibles pour plus tard. Une fois que tout ça est fait, on demande au *CtrlLogin* l'affichage de la vue qu'il gère (la fenêtre de connexion).

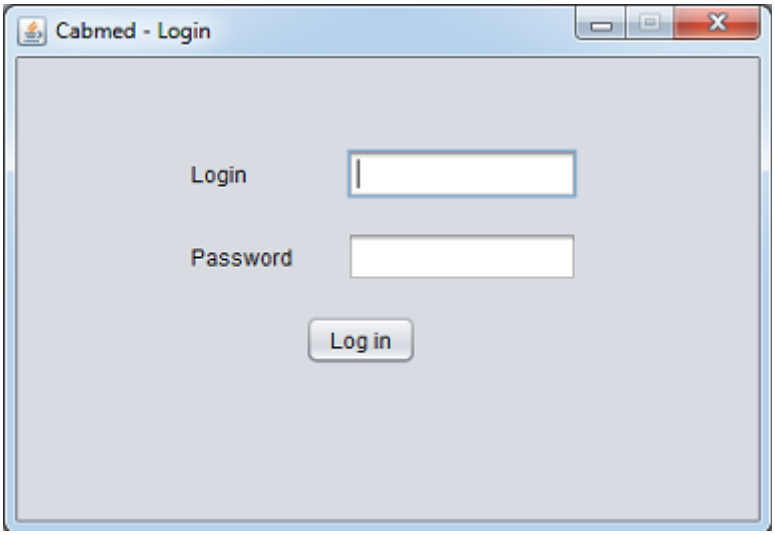
(2) – Le *CtrlLogin* est la classe qui va gérer la *VueLogin* (la fenêtre de connexion). Cette classe est instanciée par la classe *CtrlPrincipal* vue juste au-

dessus et contient une méthode "*showView()*" permettant d'afficher sa vue. Ce contrôleur ne fait pas grand chose si ce n'est vérifier (en appelant la classe *Facade* contenu dans le *CtrlPrincipal*) si les identifiants et mots de passe introduits dans la fenêtre de connexion sont correct, et si oui, demande au *CtrlPrincipal* d'afficher la *VueAdmin* via le contrôleur qui le gère (juste en-dessous).

(3) – Voilà enfin le *CtrlAdmin* qui va gérer la fenêtre principale de l'application (la *VueAdmin*) et toutes les petites fenêtres qui lui sont liées. Il y a beaucoup plus d'actions ici car dans cette application, quasi-tout passe par ce contrôleur. Tout ce qui est fait avec la base de données passe comme toujours par la classe *Facade* reçue du *CtrlPrincipal*, mais même dans ce cas-là, on passe d'abord par le *CtrlAdmin* (ce n'est jamais la vue qui appelle la *Facade*). Pour rappel, la *VueAdmin* permet de rajouter, modifier et supprimer des médecins, des secrétaires, des infirmières et des spécialisations. Ce contrôleur garde une instance de toutes les vues et chacune de ces vues a une méthode pour s'adapter à chaque appel de la vue. On verra les détails en code plus tard.

Vues : (package cabmed.admin.ihm)

(1) – La *VueLogin* est la fenêtre de connexion. Cette vue est gérée par le *CtrlLogin*. Cette vue est assez simple d'aspect et de fonctionnement. En voici un screenshot :



En dehors du code pour l'interface graphique, il n'y a que l'action pour le bouton qui est présent. Comme dit plus haut, aucune vue n'a accès à la classe *Facade* donc la vue fait la demande auprès de son contrôleur qui lui va vérifier avec la *Facade* (qui va vérifier avec la partie *DAO)* si le login est ok. Et si le login est bon, le *CtrlLogin* demande au *CtrlPrincipal* l'affichage de la *VueAdmin* (en passant donc par le *CtrlAdmin*).

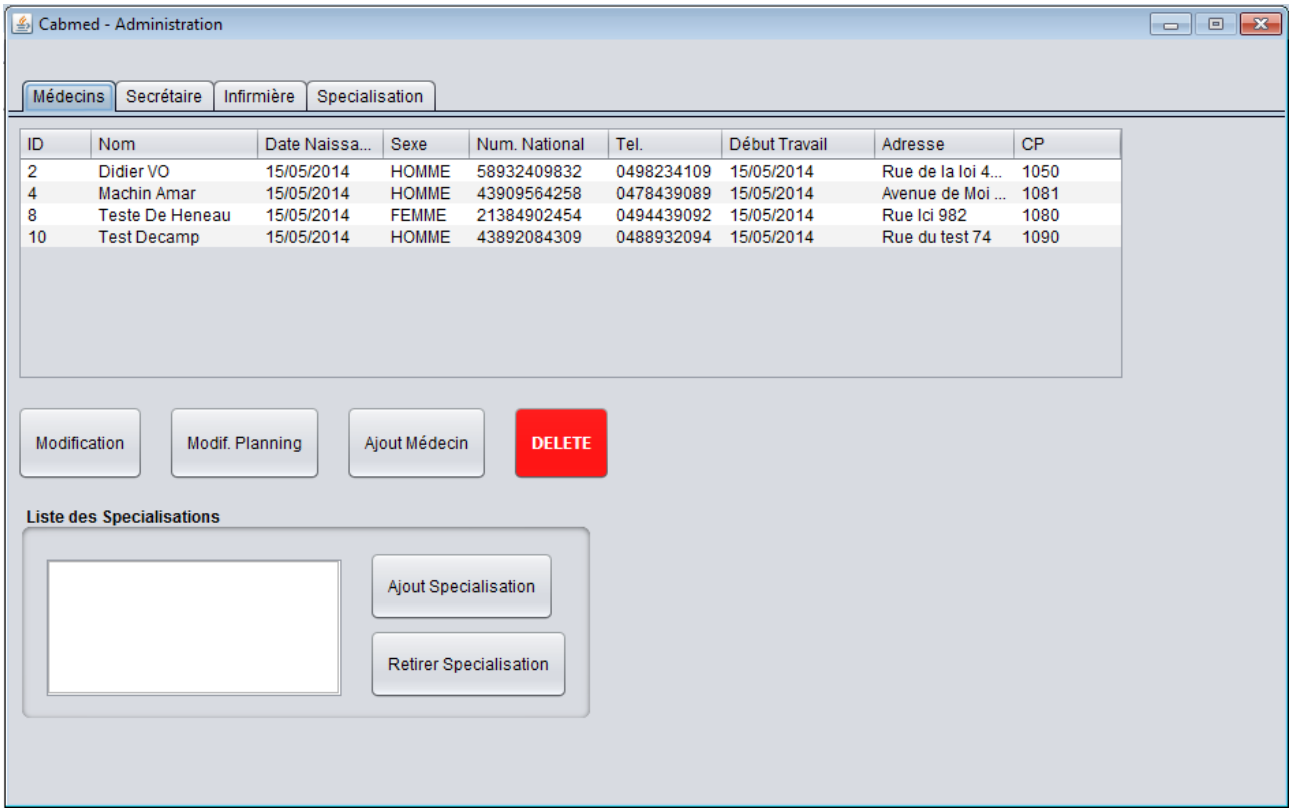
(2) – La *VueAdmin* est la vue principale de l'application, celle sur laquelle on

fait TOUT ! Cette vue est coupée en quatre parties, une partie pour la gestion de chaque élément (médecin, secrétaire, infirmière et spécialisation). Dans chacune de ces parties on retrouve un tableau reprenant toute la liste de l'élément en question (dans la partie médecin, on retrouve la liste de tous les médecins).

Commençons par la présentation de la partie gérant les médecins, et pour ça, voici un screenshot de la fenêtre :

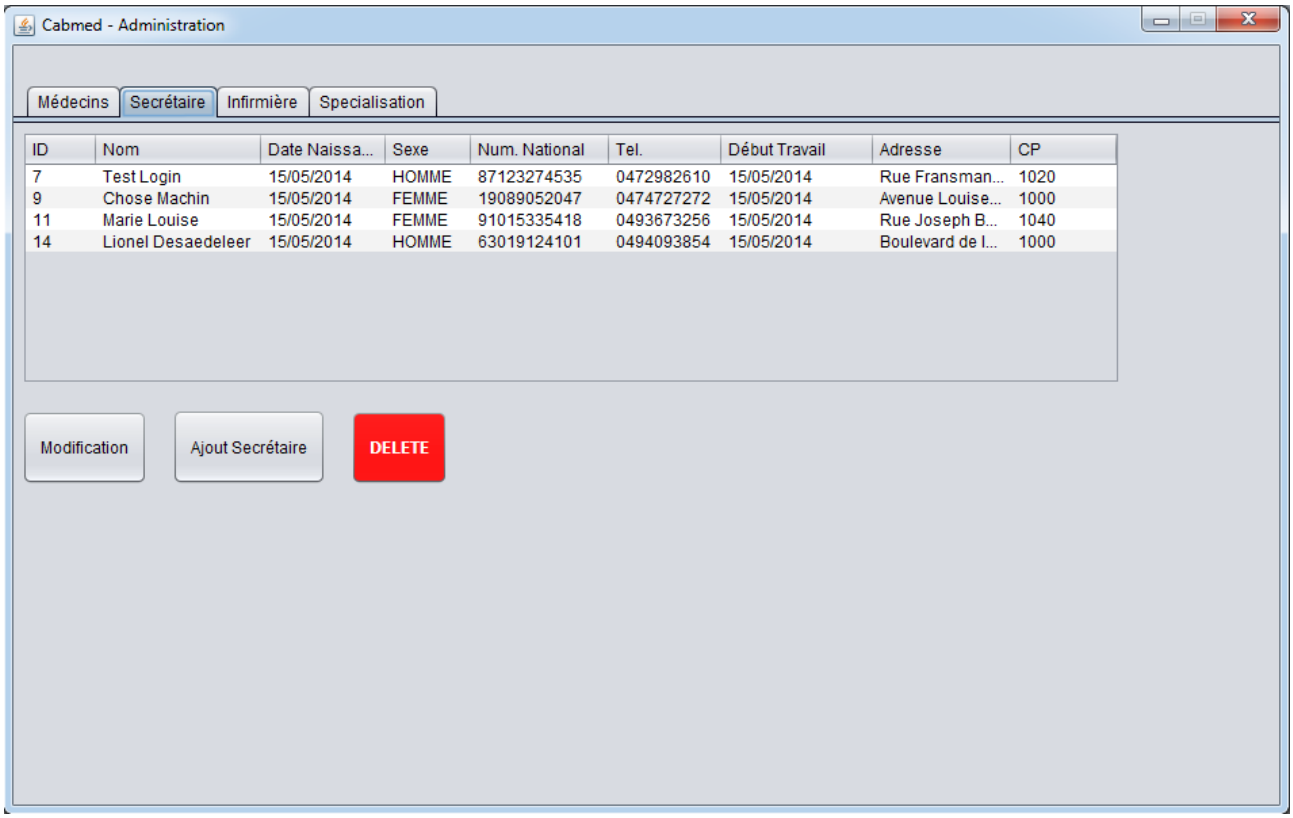
Présentation du mémoire Bachelier en Informatique de Gestion

Sujet : Cabmed – Gestion d'un cabinet médical



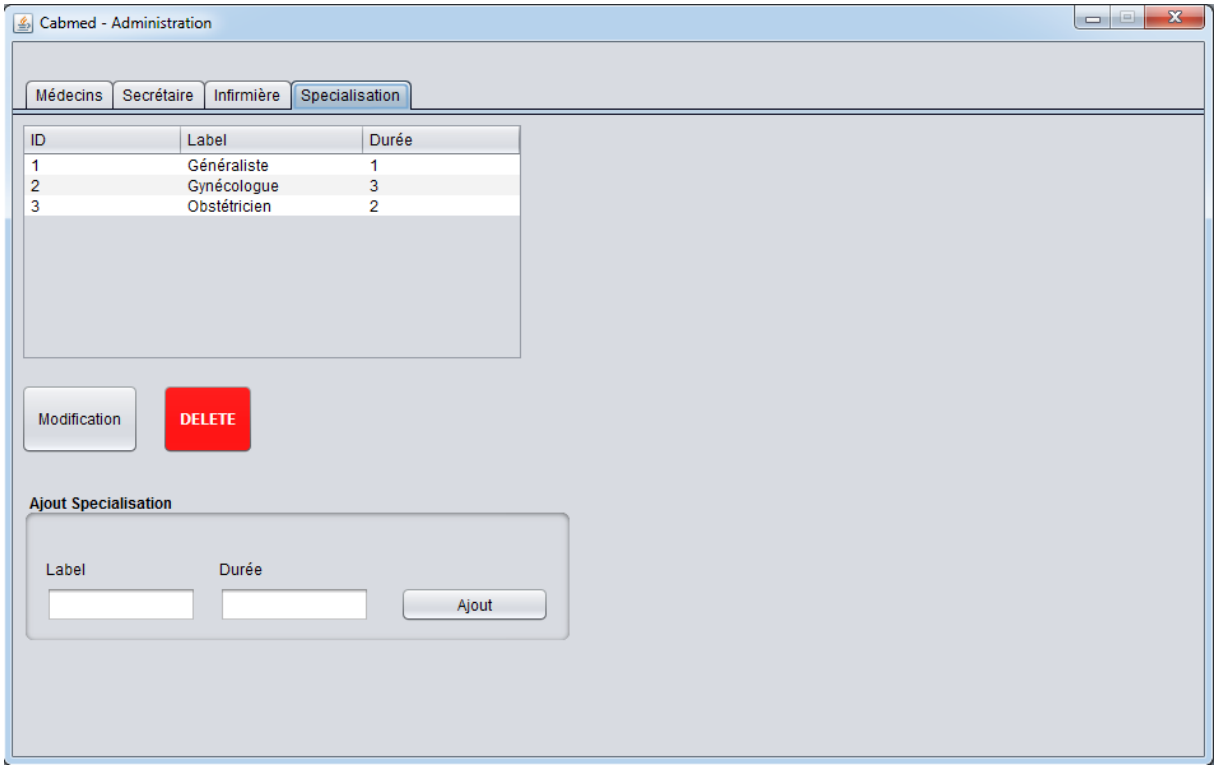
On remarque tout de suite les quatre onglets sur le dessus de la fenêtre permettant de passer d'une partie à l'autre. Je pense que l'interface graphique est assez intuitif, en sélectionnant un élément de la liste, on peut supprimer le médecin, le modifier (les informations générales concernant le médecin lui-même) ou modifier son horaire (pas ses rendez-vous mais vraiment son horaire, lui rajouter un jour de travail ou le faire commencer plus tard un certain jour de la semaine par exemple). Sur le bas de la fenêtre on voit un espace réservé aux spécialisations **à ne pas confondre avec la gestion des spécialisations dont on voit l'onglet sur le haut de la fenêtre**, ici il s'agit bien des spécialisations liés au médecin sélectionné. En sélectionnant l'un des médecins de la liste, on peut voir la liste des spécialisations qu'il a (par exemple s'il est généraliste ET obstétricien en même temps). On peut choisir de rajouter une spécialisation ou d'en supprimer une pour ce médecin. Que ce soit l'ajout de spécialisation, la modification du médecin ou de son horaire, tout se fait via des fenêtres bien spécifique dont je parlerais plus loin après avoir vu les trois autres onglets.

Voyons maintenant l'onglet réservé à la gestion des secrétaires. Comme au-dessus, commençons avec un screenshot de cette partie :



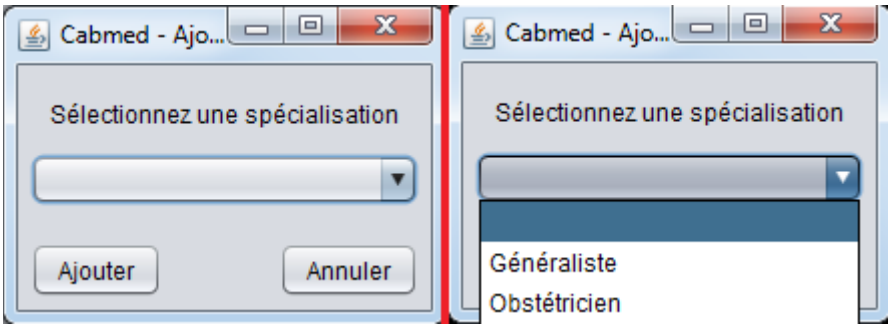
Avant de continuer je précise juste que l'on ne verra pas la partie de gestion des infirmières car elle est semblable à cette partie-là. Il y a exactement les mêmes boutons, aux mêmes endroits exécutant les mêmes actions. Ceci dit, cette vue ne diffère pas beaucoup de la vue gestion de médecins non-plus. La différence c'est qu'ici on ne gère ni horaire, ni spécialisation. On a donc les mêmes actions concernant la gestion de la personne en elle-même, c'est à dire le fait de modifier les informations générales de la secrétaire, et le fait d'en rajouter. Les fenêtres utilisées pour ça sont les mêmes que celle utilisées pour le médecin, les fenêtres s'adaptent en fonction du type d'élément choisi.

Pour terminer on verra la partie réservée à la gestion des spécialisations. Elle fonctionne elle aussi de la même manière que les autres, liste des spécialisations et actions que l'on peut exécuter sur eux. Voici un screenshot de cette partie-là :



Comme je l'ai dit, même système qu'au-dessus, les boutons avec les différentes actions que l'on peut effectuer sur chaque spécialisation. Le seul rajout par rapport aux parties précédentes c'est le panel en bas de la fenêtre. De là on peut rajouter une spécialisation. Assez intuitif aussi.

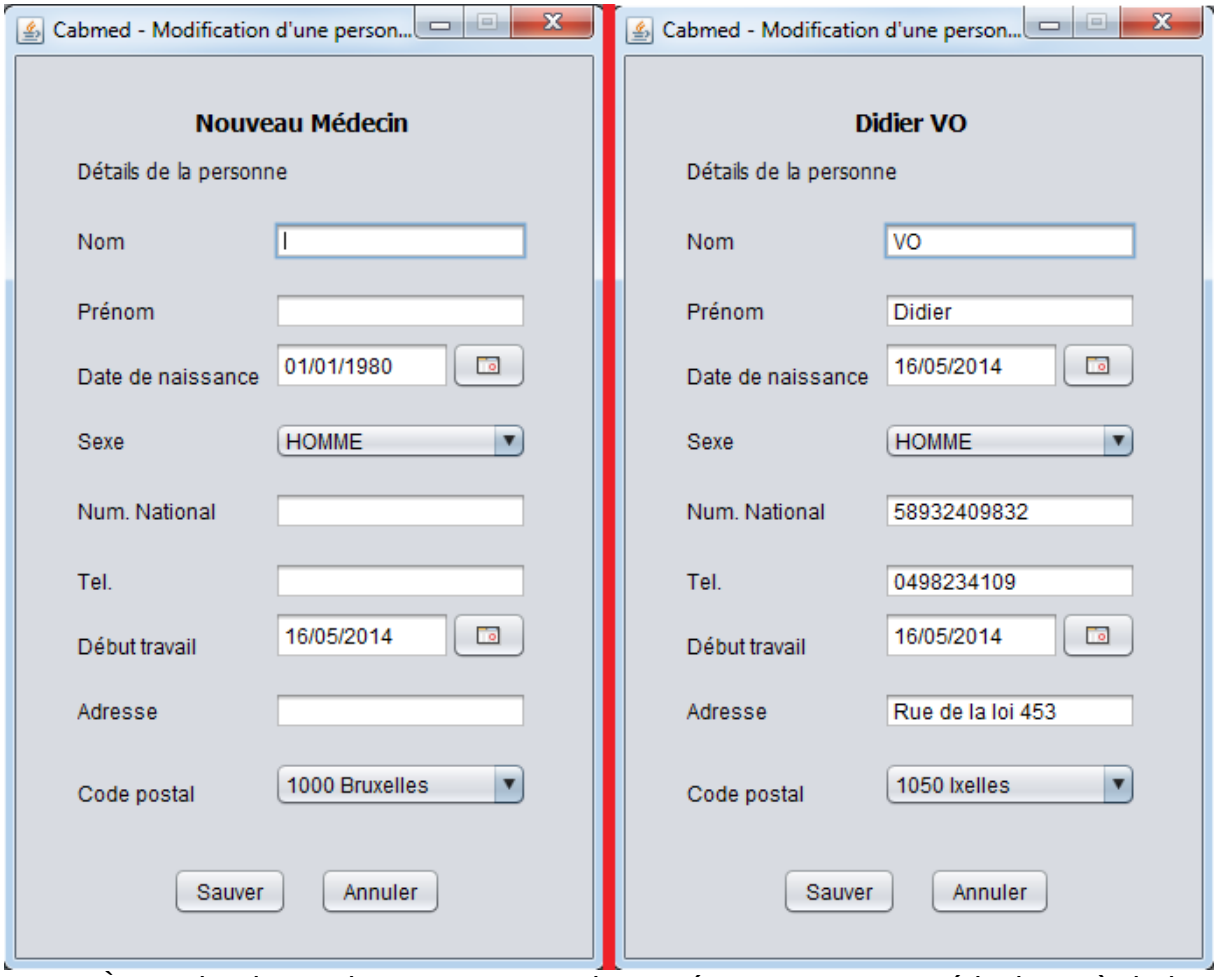
(3) – La *VueAdminAddSpecAMedecin* est la vue permettant de rajoutant une spécialisation à un médecin. Prenons l'exemple d'un médecin avec deux spécialisations comme, par exemple, généraliste et obstétricien, mais qui dans notre cabinet n'officie qu'en tant que généraliste. Un jour, d'un commun accord, nous décidons de lui rajouter sa deuxième spécialisation. Eh bien c'est ici que ça se fait. Voici un screenshot de la vue, c'est assez parlant :



À gauche, la vue à son lancement, et à droite, le menu proposant les spécialisations que le médecin sélectionné n'a pas encore (liste se basant sur toutes les spécialisations que le cabinet gère). Le filtrage de la liste se fait directement dans le *CtrlAdmin* vue plus haut. Assez simple à comprendre, on choisit la spécialisation et

on ajoute... Lorsque l'on sauvegarde, on appelle le pattern Observer pour qu'il mette les vues à jour (on verra détails plus tard).

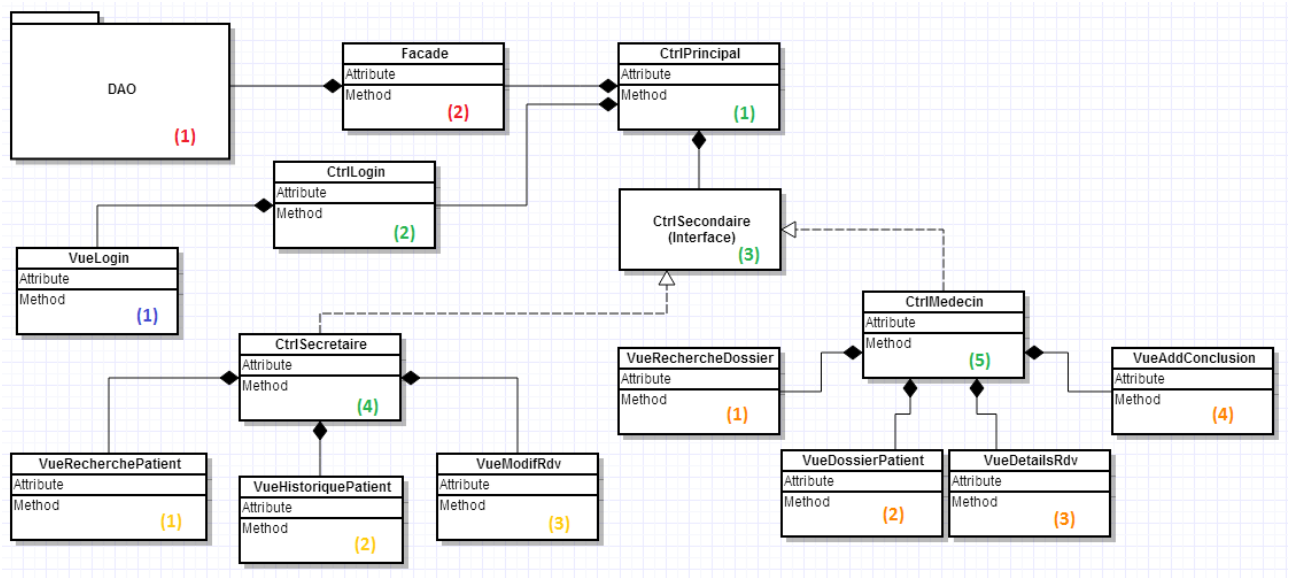
(4) – La *VueAdminModifPersonnel* est la vue nous permettant de modifier une personne, mais aussi d'en créer une nouvelle... La même vue est utilisée pour les deux actions. Elle est aussi la même que ce soit pour les médecins, les infirmières ou les secrétaires... Je l'ai faite de façon à ce qu'elle s'adapte à chaque fois qu'on l'appelle. Voici un screenshot :



À gauche, la vue lorsque nous voulons créer un nouveau médecin, et à droite, lorsqu'on veut en modifier un qui existe déjà... C'est pareil pour les secrétaires et les infirmières. Le bouton sauver appellera une méthode du *CtrlAdmin* qui servira soit à enregistrer le nouveau membre du personnel, soit à sauvegarder les modifications apportées à la personne déjà existante. On verra plus tard quel est le code qu'il y a derrière tout ça pour adapter la fenêtre à la personne choisie. Là aussi, que ce soit un ajout ou une modification de médecin, on fait appel au pattern Observer pour remettre la vue principale à jour.

(5) – La *VueAdminModifPlanning* est la vue permettant d'adapter l'horaire d'un médecin. Par défaut, lorsque l'on rajoute un médecin, il a un horaire vide

Application Manage :



Application Web :

Présentation du mémoire Bachelier en Informatique de Gestion

Sujet : Cabmed – Gestion d'un cabinet médical