Page de garde

Table des matières

1. INTRODUCTION :

1.1. Présentation du projet :

Avant de poursuivre, j'aimerais insister sur une chose : Il s'agit là de la gestion d'un **CABINET MÉDICAL**, pas d'un **HÔPITAL**.

1.2. Objectif et originalité :

1.3. Chemin emprunté et technologies choisies :

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 (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... Deux choix s'offraient à moi : implémenter ma propre gestion du planning (assez costaud à faire), ou utiliser Google Agenda qui est un outil bien assez puissant et complet pour pouvoir l'utiliser comme on le veut. L'avantage du Google Agenda c'est qu'il est assez adaptable (jusqu'à une certaine limite bien sûr...) ce qui nous permet d'en faire à peu près tout ce que l'on veut. Il y avait par contre un gros problème, c'est que je ne trouvais aucun tutoriel

(complet ou partiel) sur internet expliquant comment on pouvait utiliser Google Agenda à partir d'une application... Je sais que Google fournit des librairies (entre autre Java) permettant donc de l'utiliser à partir de n'importe où, mais nul part on trouve une personne expliquant comment utiliser ça. Je pouvais apprendre moi-même sans aucun conseil de l'extérieur mais apprendre ça m'aurait pris énormément de temps (plein de tests à faire pour être sûr de comment ça fonctionne, ...), et avec le retard que j'avais accumulé sur le mémoire, je ne pouvais pas me le permettre... J'ai donc décidé de faire ma propre gestion de planning. J'ai finalement trouvé un moyen de gérer ça moi-même mais même si c'est loin d'être aussi performant que le Google Agenda, c'est un système qui tourne bien et est plus que suffisant pour une application comme la mienne (au final, j'ai trouvé ça plus facile que ce que je pensais...).

Pour ne pas ré-inventé la roue une fois de plus (deux fois sur le même projet, c'est assez lourd...), j'ai décidé de m'orienter vers des outils existants pour certaines parties de l'application. Le premier outil utilisé est **JPA** (pour **J**ava **P**ersistence **A**pi). Cet outil est ce qu'on appelle un **ORM** (pour **O**bject **R**elational **M**apping). 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 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 ORMs pour nous. Il existe plusieurs "vrais" ORMs, 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 ORMs 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 ORMs et est donc plus rapide à mettre en place, 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 ORMs ! En apprenant à utiliser JPA, on peut utiliser tous les avantages de tous les ORMs (certains d'entre eux étant plus adaptés dans certains cas).

Le deuxième 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 IDEs 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 IDEs 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. Donc tant qu'on reste en simple Java, NetBeans est (à mes yeux) beaucoup plus puissants qu'Eclipse, par contre si on part dans de la programmation Java moins "classique" (comme la programmation Android par exemple, qui se fait aussi en Java), NetBeans ne fait pas vraiment le poids face à Eclipse...

Petits détails supplémentaires, j'ai travaillé sur mon mémoire de deux endroits différents : de chez moi et de l'endroit où je passais mon stage (pendant les moments calmes bien évidemment...). Pour pouvoir passer d'un endroit à l'autre j'utilisais ce que l'on appelle un gestionnaire de version me permettant entre autre (ce n'est pas sa fonction première...) de récupérer le code de mon projet de n'importe où. Un gestionnaire de version utilise un dossier (appelé un repository) pour stocker le projet, ce dossier peut être sur son propre ordinateur ou quelque part sur internet, et grâce à ça j'ai pu travaillé sur mon projet de ces deux endroits.

1.4. Informations techniques :

Quelques détails techniques qu'il pourrait être intéressants de connaître :

- **Version de Java :** Java SE (orienté pc) & EE (orienté web) 7
- **Environnement de développement :** NetBeans 7.4
- **Base de données :** MySQL 5.6
- **Serveur d'application web Java :** Glassfish Server 4.0
- **Gestionnaire de version :** Git avec le site "github.com" comme repository.

1.5. À voir dans ce mémoire :

Dans cet écrit nous verrons plusieurs choses. Je commencerais par parler de l'analyse faite avant de me lancer dans l'application, expliquant les recherches effectuées ainsi que quelques exemples de cas d'utilisations. Je continuerais en parlant de l'implémentation du projet en lui-même, quelle est la structure des différentes sous-projets avec des exemples de code où les différents ils interagissent. En effet, chacune des applications est un projet à part entière, mais pas seulement ! Le dernier chapitre sera réservé à différentes choses que je trouve intéressantes, que ce soit les différents **patterns** utilisés (où, pourquoi et comment?) ou quelques bouts de code que je trouve intéressants de revoir (comme justement le planning qui était le plus gros problème qui s'est présenté dans ce projet, comment je l'ai géré en fin de compte?).

2. ANALYSE :

2.1. Recherche effectuée :

Concernant la recherche liée à ce mémoire, j'avais plusieurs choses de prévues qui en fin de compte ont fini par tomber à l'eau... La principale source d'informations que je pouvais avoir était un cabinet médical se trouvant près de chez moi. Le responsable de ce cabinet tient à côté une pharmacie depuis des lustres, le cabinet date de seulement quelques années. Néanmoins, le connaissant depuis pas mal de temps déjà de par sa pharmacie, je pensais aller lui demander des informations concernant la gestion d'un cabinet médical. Chanceux comme je suis, avant que je prenne le temps d'y aller (très long stage imprévu de 4 mois que je passais en même temps...), le cabinet a fermé pour travaux...

J'ai du laisser tomber ma source d'informations principales et ai du m'orienter vers autre chose. Je recherchais sur internet ce que l'on pouvait trouver sur l'analyse d'un cabinet médical. Pas une analyse toute faite non plus, mais de petites informations : une discussion dans un forum d'une personne demandant ce qu'elle aurait besoin, ... Donc que des bribes d'informations, pour que je me fasse mon analyse moi-même ! Et c'est tout ce que j'ai trouvé, des bribes d'informations ! La seule "grosse" source d'informations que j'ai pu avoir étaient une application déjà existante : Medimust est une application du même genre que celle que je veux faire. Sur leur site, la société à qui appartient le logiciel la présente sous tous ses aspects. C'était parfait, grâce à ça je pouvais voir ce qu'il fallait encore gérer et que je n'avais pas prévu... Je n'ai pas essayé de reproduire l'application entière, mais j'ai gardé quelques idées d'elle.

2.2. Cas d'utilisation :

Ici je ne présenterais pas tous les use case faits pour ce projet, ça risque d'être beaucoup trop lourd. Je préfère n'en présenter que deux, un assez simple et l'autre un peu plus complexe.

a) Use Case Login : (Application Web, Admin et Manage)

Pré-requis:

- La personne voulant se connecter doit déjà être enregistrée

Happy Path:

1- La personne arrive sur la page/fenêtre de connexion

2- La personne introduit son e-mail (son login) et son mot de passe

3- L'utilisateur est redirigé vers une page/fenêtre qui varie selon la fonction de l'utilisateur.

Cas particuliers:

<u>1) Login impossible au point 2 du Happy Path (sur application Web)</u>

- Si le login ne passe pas (login ou mot de passe incorrect), la personne aura toujours la possibilité de choisir l'option mot de passe oublié

- Un mail lui sera alors envoyé pour la validation

- Dans le mail, redirection vers une page proposant de choisir un nouveau mot de passe (à entrer deux fois)

- Une fois validé, la personne sera redirigée à nouveau vers la page login

<u>2) Login impossible au point 2 du Happy Path (personnel sur application Admin ou Manage)</u>

- La personne devra contacter l'administrateur

b) Use Case Enregistrer/Rechercher un patient: (Application Manage)

Pré-requis:

- La secrétaire qui fait l'enregistrement/la recherche doit être loggée

Happy Path:

1- Un patient se présente devant la secrétaire du cabinet

2- La secrétaire prend la carte d'identité du patient et l'insère dans son lecteur de carte

3- Sur l'application, la secrétaire lance la lecture de la carte

4- Au chargement, l'application vérifie dans la DB que ce numéro de registre national n'existe pas déjà.

5- Une fois toutes les infos chargées (de la DB ou de la carte, sera précisé sur l'application), la secrétaire peut modifier certaines choses avant d'enregistrer

6- Une fois sauvegardé, la secrétaire pourra passer directement à la prise de RDV

Cas particuliers:

<u>1) Au point 3 du Happy Path, la lecture de la carte peut échouer (problème matériel ou logiciel)</u>

- La secrétaire doit signaler l'échec mais en attentdant elle peut choisir de rechercher/d'enregistrer manuellement un patient

- Il lui sera d'abord demandé le numéro de registre national, l'application vérifiera si la patient éxiste déjà.

  - Si le patient éxiste déjà, les infos seront chargées

  - Si le patient n'existe pas, la secrétaire devra rentrer le reste des informations

- On reprend au point 5 du Happy Path

2.3. "Installation" et scénario:

Ici j'aimerais parler du "scénario" que j'envisage... Ce que j'appelle scénario c'est comment se passerait la "vie" du cabinet médical. Qui fait quoi ? Comment ? ... C'est important pour bien comprendre le fonctionnement des applications développées. Pour ça je commencerais par la toute première installation des applications, je continuerais avec l'enregistrement d'un patient par une secrétaire pour finir avec la clôture de son rendez-vous par le médecin qui l'aura ausculté. Je parle du fonctionnement de l'application mais sans trop de détails. Pour ça il faut aller voir le chapitre sur la structure des différents projets où j'expose la liste de toutes les classes utilisées ainsi que des screenshots de chaque vues de l'application.

Je parle "d'installation" un peu à tord dans le sens où les applications ne nécessitent pas d'être installées pour êtres utilisées. Par contre elles peuvent avoir besoin de certaines choses pour fonctionner, et c'est bien ça que j'ai en tête quand je parle d'installation.

Pour la partie installation je suppose que le cabinet médical possède déjà les serveurs nécessaires et que tout soit déjà implémenté dessus (serveur de base de données MySQL avec la base de données déjà créée, et serveur d'application web Java avec l'application web déjà déployée). Je laisse ces deux points de côté étant donné leur nature "variable". En effet, créer une base de données sur un serveur proposé par OVH par exemple se fait d'une certaine manière (en passant par PHPMyAdmin seulement), alors que d'autres fournisseurs du même service pourrais faire ça différemment. C'est pareil pour l'application web dont le déploiement varie en fonction du serveur d'application choisi. Je préfère donc rester sur du simple pour l'instant et ne parler que des applications orientées pc : les applications Admin et Manage. Petit détail supplémentaire, comme dit plus tôt, il n'existera qu'**UN SEUL** administrateur ! Étant donné qu'il s'agisse d'un cabinet médical (et pas d'un hôpital...), je par du principe qu'il n'y a qu'un seul gestionnaire du personnel. Donc l'identifiant et mot de passe du **SEUL** administrateur seront enregistrés dés la création de la base de données et seront donc déjà disponibles lors de l'installation des applications.

Installation des applications :

Le plus simple est de commencer par l'installation de l'application Admin pour que l'on puisse déjà créer toutes les données nécessaires à l'utilisation de l'application Manage. Donc comme dit plus haut, pas d'installation nécessaire : une fois le fichier copié dans le pc, il nous suffit de le lancer... Une fois dessus il nous demande l'identifiant et le mot de passe de l'administrateur. Une fois introduits, l'application nous redirige vers la page nous permettant de gérer le personnel et les spécialisations. Ici nous rajoutons donc toutes les spécialisations que le cabinet doit pouvoir gérer ainsi que tout le personnel nécessaire avec, dans le cas des médecins, leur disponibilité et leur(s) spécialisation(s). Une fois tout cela fait, le cabinet est prêt à "vivre"...

Passons maintenant à l'application Manage. Ici c'est la même chose, pas d'installation pour l'application elle-même, par contre, nous avons besoin d'une chose en plus dans le cas de la secrétaire, c'est le pilote nécessaire à l'utilisation du lecteur de carte. L'application peut fonctionner sans, car la secrétaire peut se passer de cette option pour retrouver ou enregistrer un patient. Inutile donc d'installer le pilote sur tous les ordinateurs du cabinet, juste ceux qui pourraient être utilisés par une secrétaire. Une fois le pilote installé, l'application est complète ! En dehors des installations, il faut bien évidemment que les ordinateurs soient connectés à Internet pour pouvoir accéder à la base de données... Sans ça, l'application pourra se lancer mais sans pouvoir rien faire...

Nous voici donc dans la situation où toutes les applications sont installées et fonctionnelles ! Il ne reste plus qu'à les utiliser. Passons donc à la partie "scénario" où j'explique en gros comment utiliser l'application Manage. Pour ce qui est de l'application Admin, c'est presque complètement expliqué plus haut, il suffit de rajouter les spécialisations, et le personnel... Plus de d'infos dans les détails de chaque projet plus loin.

Un patient se présente à l'accueil :

Au début de sa journée, la secrétaire lance l'application et se connecte avec ses identifiants. Elle tombe sur la fenêtre lui permettant de retrouver un patient.

Un patient se présente devant la secrétaire. Ce patient n'est encore jamais venu dans ce cabinet et présente donc sa carte d'identité à la secrétaire. Cette dernière introduit la carte d'identité dans son lecteur de carte et lance la lecture de la carte. L'application vérifie d'abord si le patient existe déjà dans la base de données avec son numéro de registre national. Si le patient existe, l'application charge toutes ses informations venant de la base de données, s'il n'existe pas, l'application charge les autres données du patient à partir de la carte d'identité. La secrétaire a la possibilité de modifier ou non les données chargées avant de les enregistrer. Une fois cette étape passée, la secrétaire est redirigée vers la fenêtre lui indiquant toutes les informations de ce patient ainsi que l'historique de ces rendez-vous actuels et passés avec la possibilité de modifier les rendez-vous actuels. La secrétaire rajoute donc un rendez-vous à ce patient en sélectionnant le médecin ainsi que la spécialisation pour laquelle le patient se présente (s'il vient pour une consultation générale ou s'il vient voir pour un problème concernant un domaine en particulier).

Une fois qu'arrive la date du rendez-vous, le patient se présente donc au cabinet pour sa consultation. Il se présente à l'accueil qui confirme bien son rendez-vous et prévient donc le médecin du fait que le patient est là.

Le médecin le reçoit et retrouve le dossier du patient avec le numéro du rendez-vous. Une fois retrouvé, le médecin ausculte donc son patient et clôture son rendez-vous en rajoutant une remarque et/ou des prescriptions à son dossier. Le médecin a la possibilité d'imprimer les prescriptions pour les donner au patient.

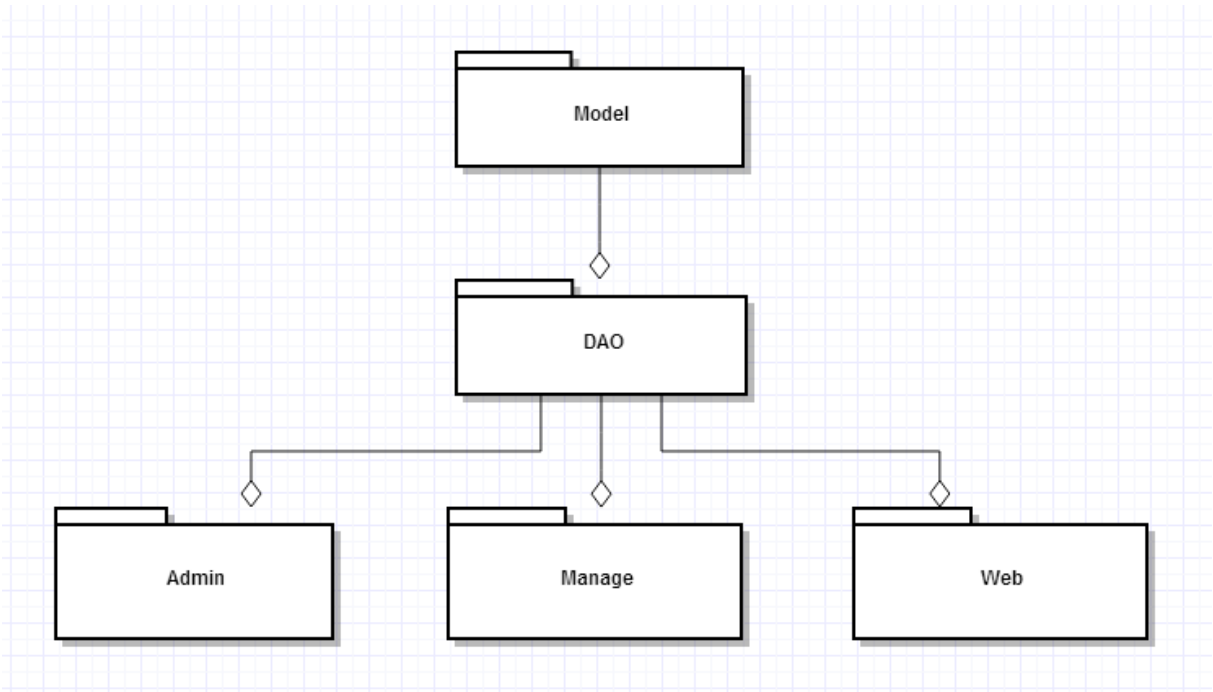
Voici donc la fin du "scénario", on est passé par tous les aspects de l'application sans non plus voir les détails. Comme dit plus haut, ça c'est dans le chapitre Implémentation qui suit, on y retrouve tous les détails possibles. Mais au moins maintenant vous connaissez un peu plus le projet et pourrez plus facilement comprendre le reste de ce que je présenterais dans cet écrit.

3. IMPLÉMENTATION DU PROJET :

3.1. Structure des sous-projets :

a) Structure des projets

Comme dit plus haut, le projet est divisé en trois applications, qui dans la pratique se divisent aussi en trois sous-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 :



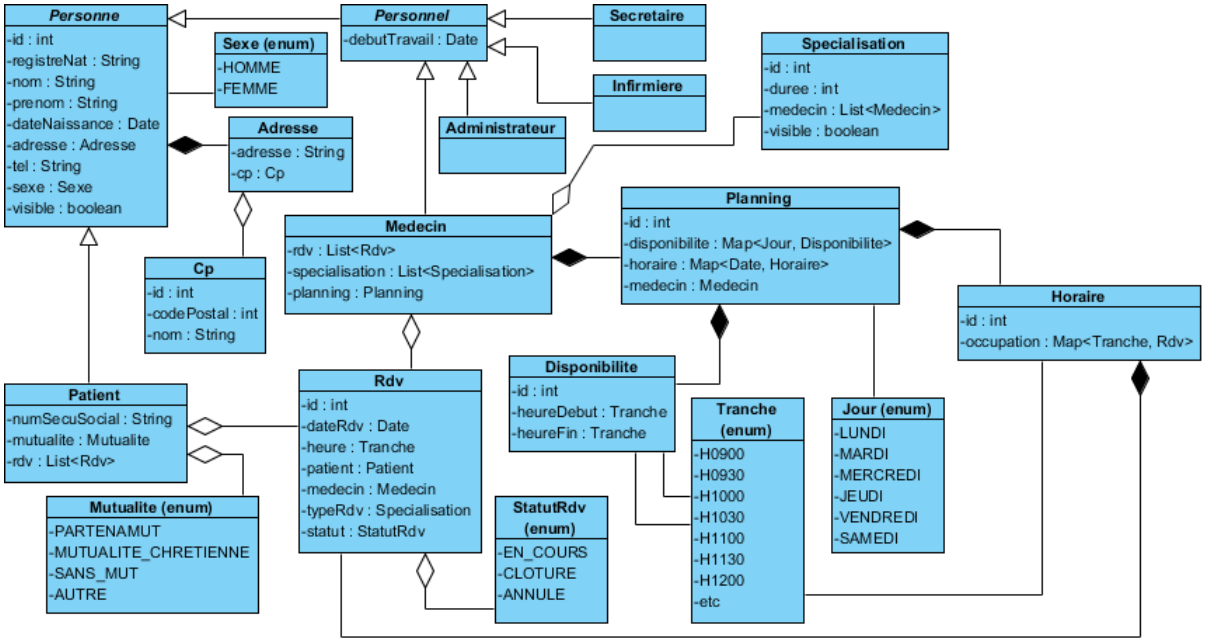
Ce qu'il faut comprendre ici c'est que chaque "dossier" (en Java on parle plutôt de "package") représente un projet spécifique. Le lien qui lie par exemple les projets *Model* et *DAO* signifie que le projet *Model* a été inséré dans le projet *DAO* (les objets servant à gérer la base de données ayant 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…

b) Model

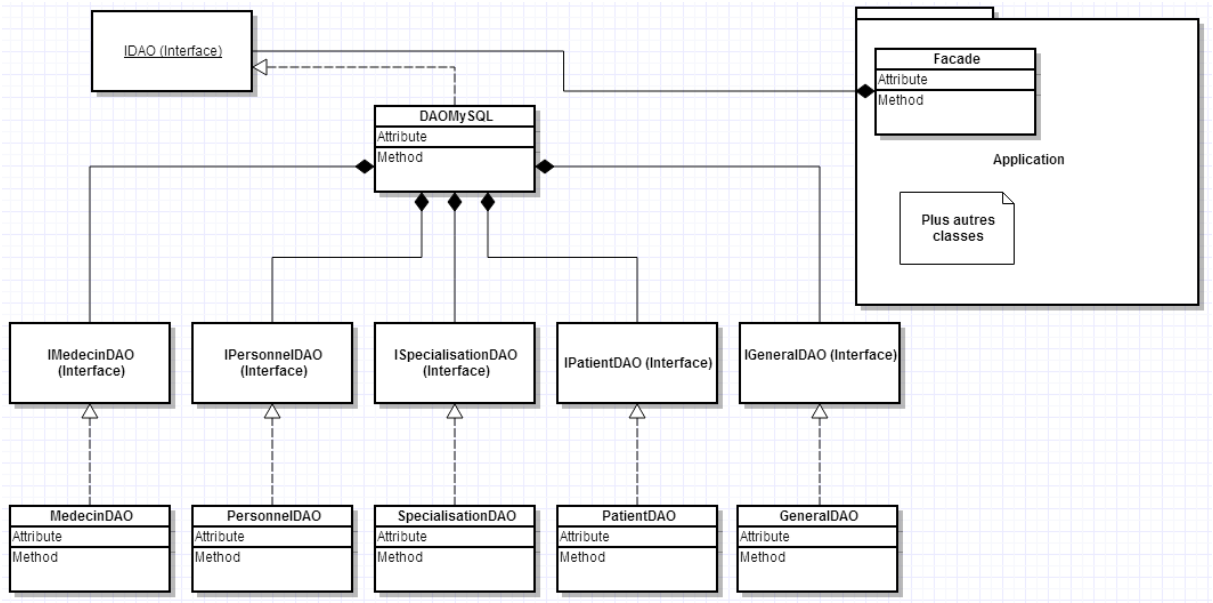
Abc



Abc

c) DAO

Abc

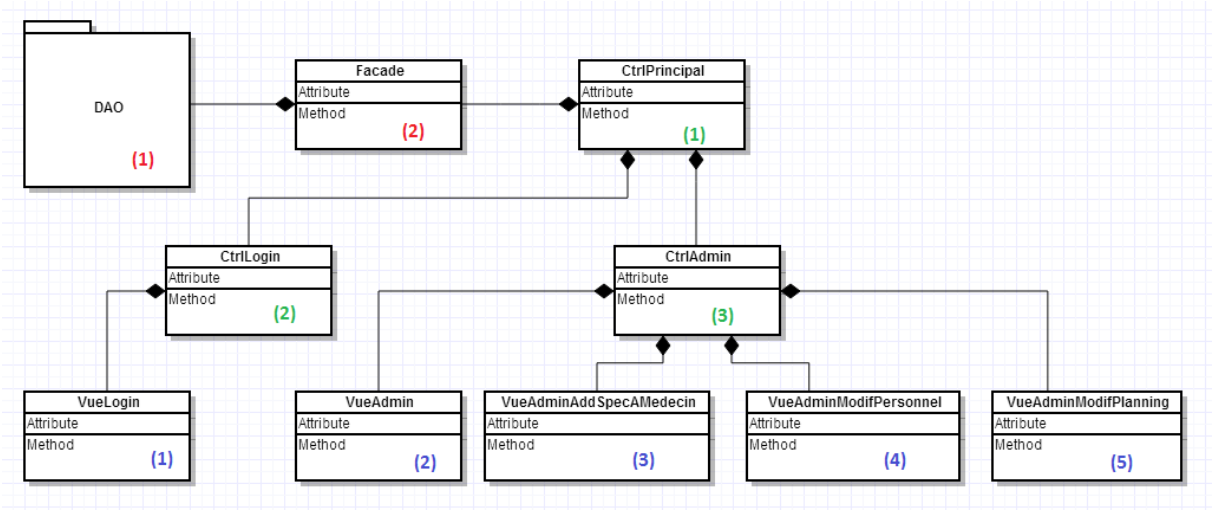

IDAO (Interface)
DAOMySQL
Attribute
Method
Facade
Attribute
Method
Application
Plus autres classes
IMedecinDAO (Interface)
IPersonnelDAO (Interface)
ISpecialisationDAO (Interface)
IPatientDAO (Interface)
IGeneralDAO (Interface)
MedecinDAO
Attribute
Method
PersonnelDAO
Attribute
Method
SpecialisationDAO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

Abc

d) 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).

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...). Plus d'infos dans le chapitre réservé aux patterns.

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).

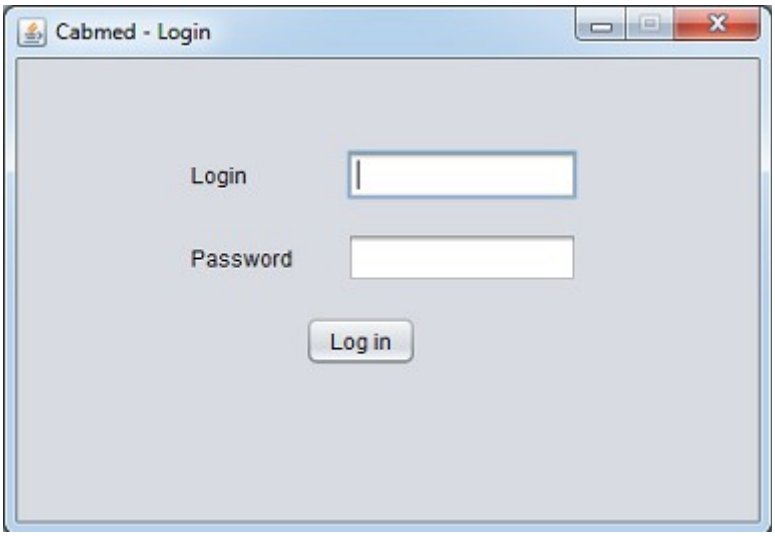
(2) – Le *CtrlLogin* est la classe qui va gérer la *VueLogin* (la fenêtre de connexion). Cette classe est instanciée par la classe *CtrlPrincipal* vue juste au-dessus et contient une méthode "*showView()*" permettant d'afficher sa vue. Ce contrôleur ne fait pas grand chose si ce n'est vérifier (en appelant la classe *Facade* contenu dans le *CtrlPrincipal*) si l'identifiant et le mot de passe introduits dans la fenêtre de connexion sont corrects, 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

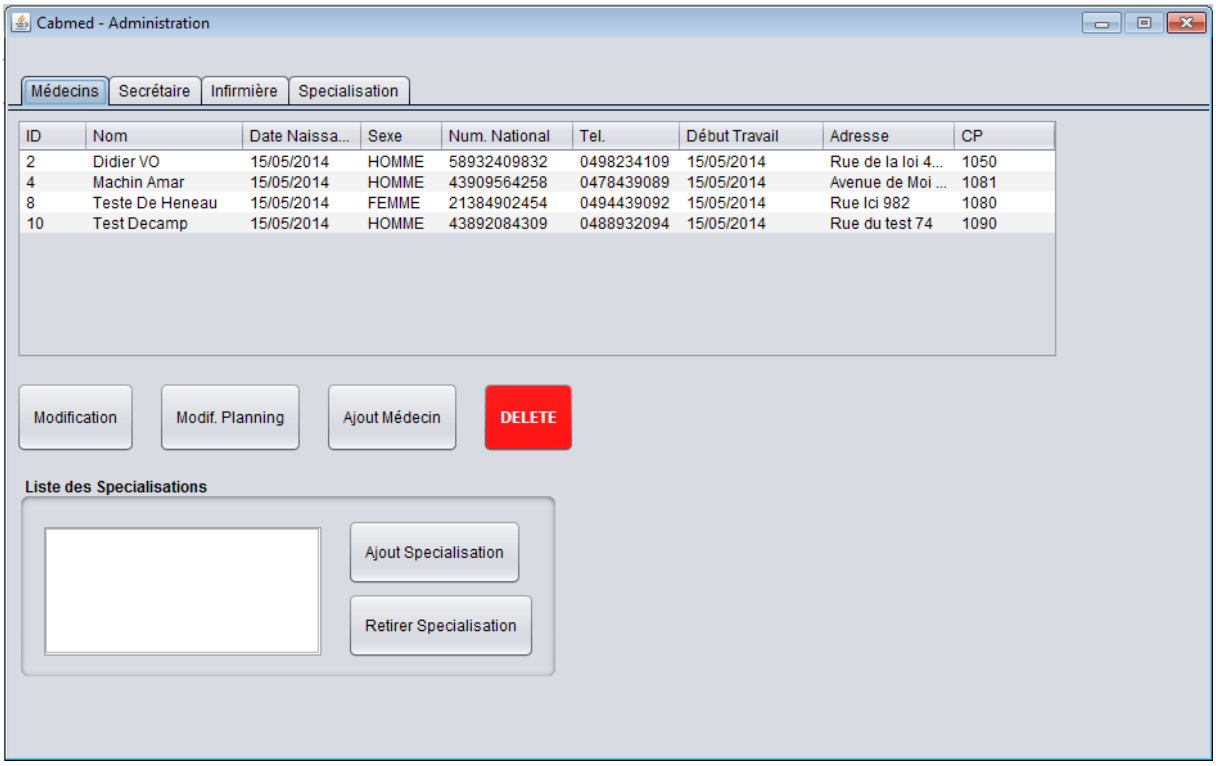
fonctionnement. En voici un screenshot :



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*).

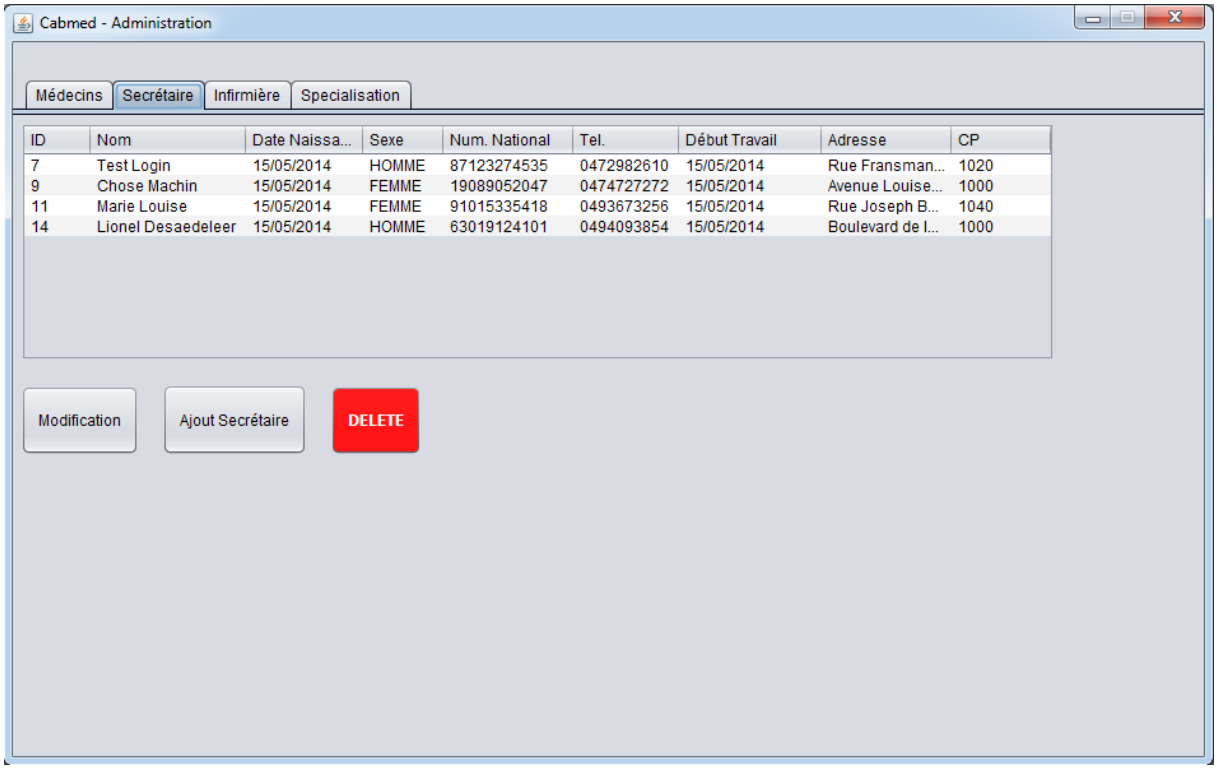
(2) – La *VueAdmin* est la vue principale de l'application, celle sur laquelle on fait TOUT ! Cette vue est coupée en quatre parties, une partie pour la gestion de chaque élément (médecin, secrétaire, infirmière et spécialisation). Dans chacune de ces parties on retrouve un tableau reprenant toute la 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 :



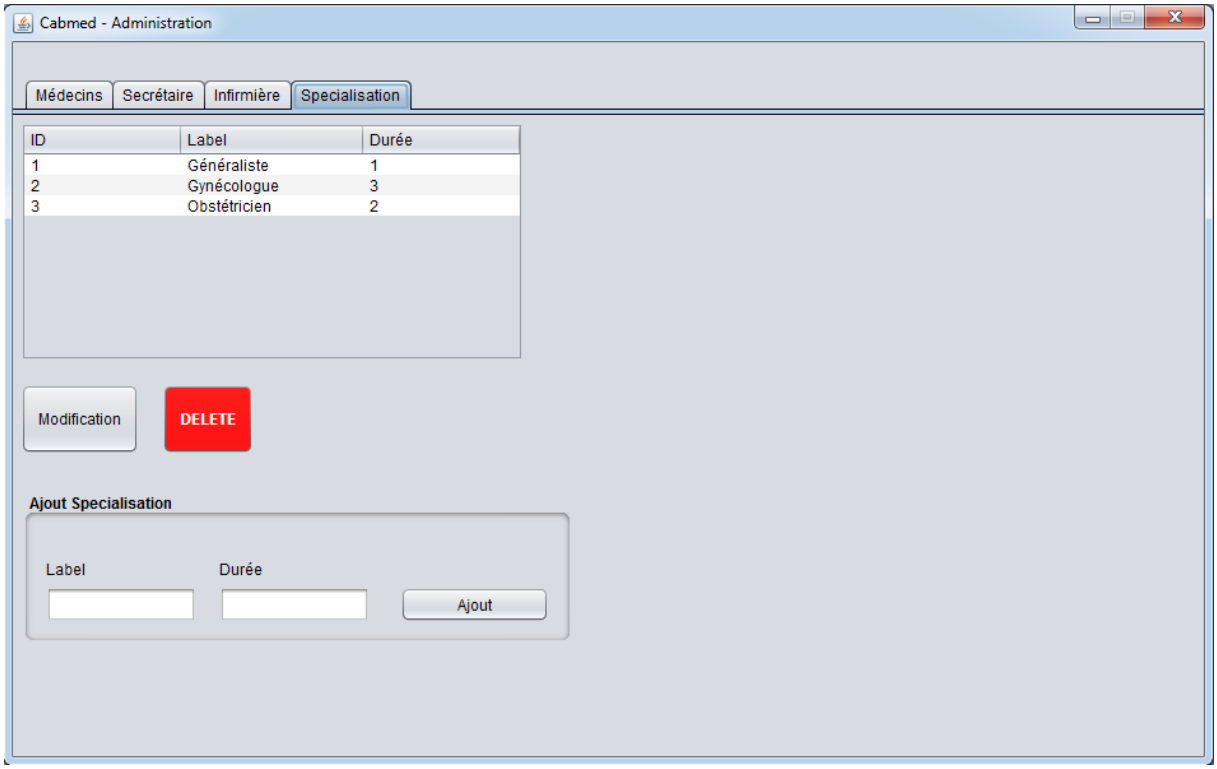
On remarque tout de suite les quatre onglets sur le dessus de la fenêtre permettant de passer d'une partie à l'autre. Je pense que l'interface graphique est assez intuitive, en sélectionnant un élément de la liste, on peut supprimer le médecin, le modifier (les informations générales concernant le médecin lui-même) ou modifier ses disponibilités (pas ses rendez-vous mais vraiment ses disponibilités, 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 ses disponibilités, 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 :



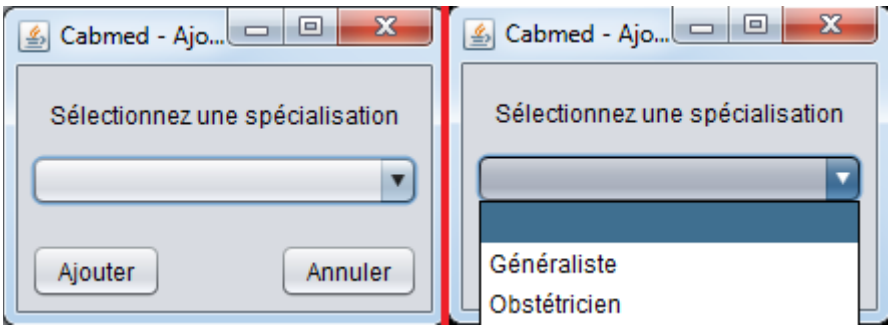
Avant de continuer je précise juste que l'on ne verra pas la partie de gestion des infirmières car elle est semblable à cette partie-là. Il y a exactement les mêmes boutons, aux mêmes endroits exécutant les mêmes actions. Ceci dit, cette vue ne diffère pas beaucoup de la vue gestion de médecins non-plus. La différence c'est qu'ici on ne gère ni disponibilités, 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 :



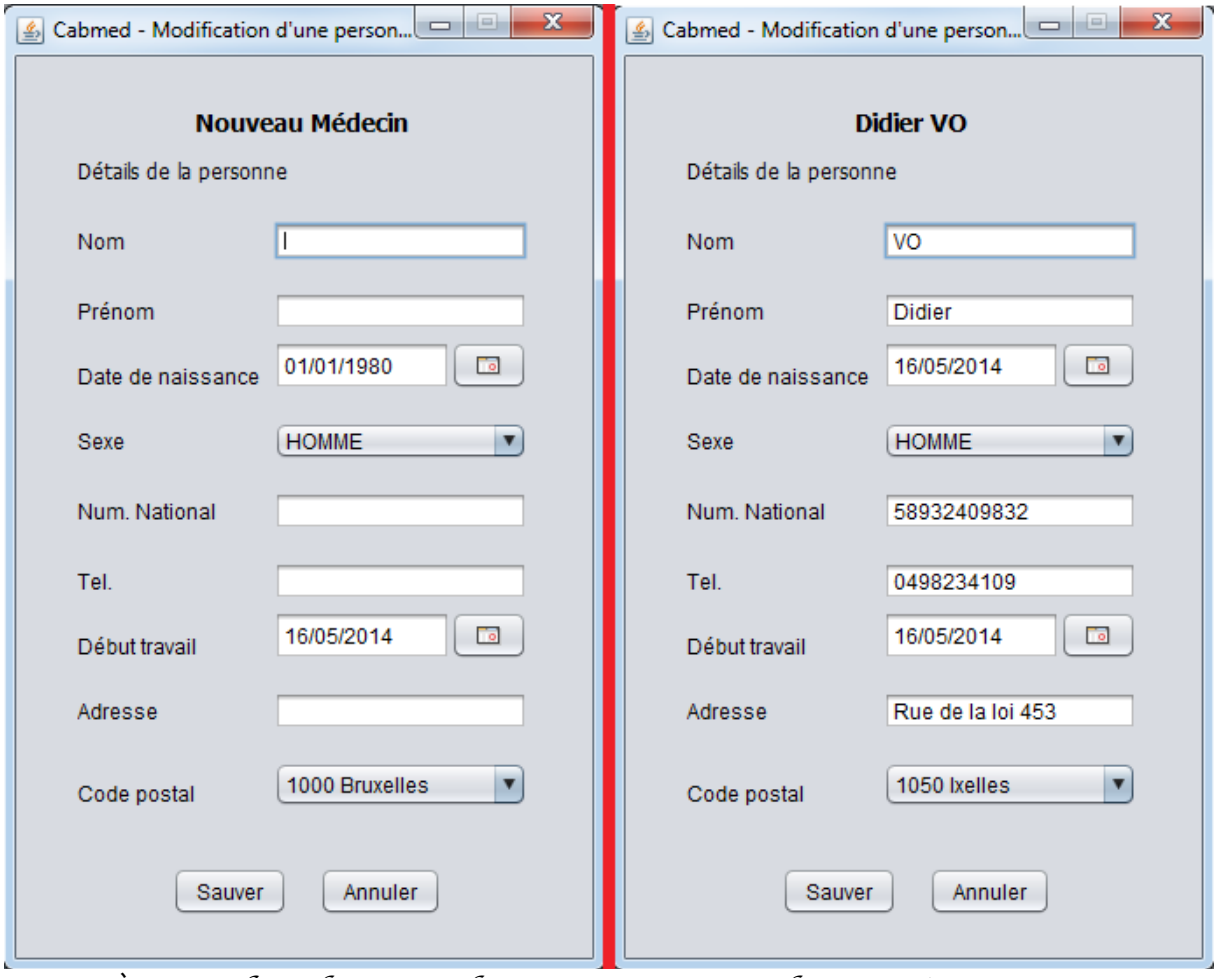
Comme je l'ai dit, même système qu'au-dessus, les boutons avec les différentes actions que l'on peut effectuer sur chaque spécialisation. Le seul rajout par rapport aux parties précédentes c'est le panel en bas de la fenêtre. De là on peut rajouter une spécialisation. Assez intuitif aussi.

(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 les disponibilités d'un médecin. Par défaut, lorsque l'on rajoute un médecin, il a un horaire vide. On doit obligatoirement passer par cette option pour lui donner un horaire (savoir quand il travaille et entre quelles heures). Souvent les médecins ont un horaire variables (que ce soit dans un cabinet médical ou un hôpital), c'est pourquoi j'ai voulu implémenter ça pour le médecin mais seulement pour lui, pas pour les infirmières ou les secrétaires, qui ont habituellement un horaire fixe... Aussi le médecin est le seul à avoir des rendez-vous, il est donc le seul pour lequel l'application a besoin de connaître l'horaire. Voici encore une fois le screenshot de cette fenêtre :

Screenshot VueModifPlanning

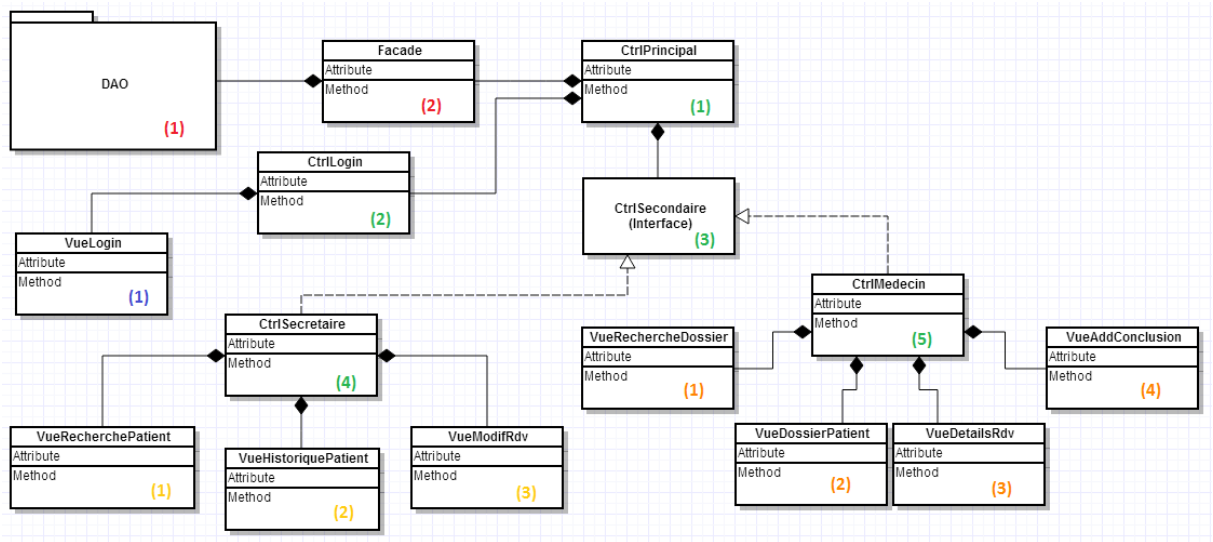
On peut adapter les disponibilités d'un médecin pour chaque jour. Il y a donc une colonne pour chaque jour de travail. On peut ne rien mettre pour certains jours, ce sera donc des jours de congé pour ce médecin.

Voici pour l'application Admin, tout ce qu'il y avait à dire a été dit ! Passons maintenant à l'application Manage, celle utilisée par les secrétaires et médecins.

e) Application Manage

Nous voici enfin à l'application Manage ! Cette application est celle utilisée par les secrétaires et les médecins. L'application s'adapte selon la personne qui se connecte. Si l'utilisateur est une secrétaire, on est redirigé vers la partie de l'application permettant de rechercher des patients et de leur fixer des rendez-vous. On a aussi accès à l'historique des rendez-vous de ce patient mais sans détails (juste le médecin, la date et l'heure ainsi que la spécialisation). Si par contre on se connecte en tant que médecin, on arrive sur une fenêtre permettant de retrouver un rendez-vous en se basant sur un identifiant (qui sera la clé primaire du rendez-vous). Une fois le rendez-vous trouvé, le médecin est redirigé sur la fenêtre permettant de gérer ce rendez-vous sur laquelle sont affichées les informations de ce patient (les noms, prénoms, ...), les généralités (ce patient prend tel médicament régulièrement, a telle maladie, ...) et permet de rajouter une remarque et des prescriptions pour ce rendez-vous. Ici aussi on a l'historique des rendez-vous, mais avec tous les détails comme la remarque qu'a laissé le médecin ainsi que les différentes prescriptions qu'a laissés ce médecin.

Comme pour l'application Admin, voici le schéma de l'application reprenant toutes les classes avec leurs explications plus bas :



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.manage.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...). Plus d'infos dans le chapitre réservé aux patterns.

Contrôleurs : (package cabmed.manage.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 fenêtre suivante en fonction du type de la personne qui s'est connecté via le contrôleur qui le gère (juste en-dessous).

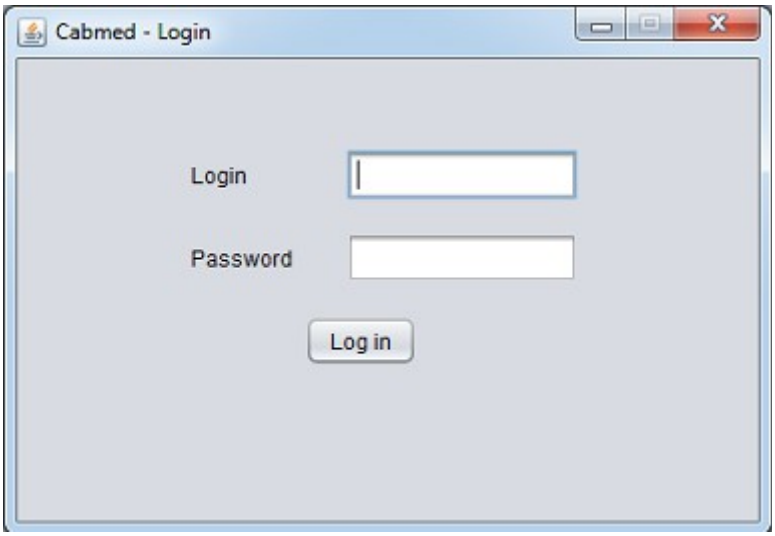
(3) – Le CtrlSecondaire est une interface implémenté par deux classes (les deux contrôleurs qui suivent). Comme dit juste au-dessus, l'application affichera des fenêtres différentes selon la personne qui s'est connectée. Je voulais faire en sorte que le CtrlPrincipal n'ait qu'une seule instance de contrôleur (en dehors du CtrlLogin) qui soit appelée quel que soit son type. J'ai donc fait une interface que je stocke dans le CtrlPrincipal et qui sera instanciée après la connexion de l'utilisateur. Si c'est une secrétaire qui se connecte, le CtrlPrincpal instanciera le CtrlSecretaire pour l'attribut de type CtrlSecondaire, si c'est un médecin qui se connecte, le CtrlPrincipal instanciera donc le CtrlMedecin, et dans les deux cas, le CtrlPrincipal lancera la méthode "showView()" de ce contrôleur qui implémentera la méthode à sa façon (le CtrlSecretaire affichera les vues destinées aux secrétaires, et le CtrlMedecin, celles destinées aux médecins).

(4) – Voici donc le CtrlSecretaire qui s'occupe de toute la partie réservée aux secrétaires. La vue instanciée par défaut par ce contrôleur est la VueRecherchePatient. Le nom est assez clair, c'est la vue qui nous permettra de retrouver un patient par son numéro de registre national (détails à la présentation de la vue). C'est donc la vue principale (lancée au départ) mais pas la seule vue gérée par ce contrôleur. En effet, la vue VueHistoriquePatient (fenêtre affichée une fois le patient trouvé) et la vue VueModifRdv (assez parlant...) sont aussi gérées par ce même contrôleur et pour respecter le pattern MVC, chaque vue passe par le contrôleur pour en appeler une autre (plus de détails dans la partie réservée aux patterns).

(5) – Nous voilà enfin au CtrlMedecin qui est donc le contrôleur qui gère toute la partie réservée aux médecins. Ici la vue qui s'affiche au lancement est la vue VueRechercheDossier. Ici on retrouve le dossier d'un patient par le numéro de rendez-vous (là encore, détails plus loin). Ce contrôleur aussi a plusieurs vues à gérer : la vue VueDossierPatient (une fois le dossier du patient retrouvé), la vue VueDetailsRdv (vue permettant d'afficher les détails d'un ancien rendez-vous après l'avoir sélectionné) et enfin la vue VueAddConclusion (la vue sur laquelle le médecin rajoute sa conclusion, d'éventuelles prescriptions et clôture le rendez-vous en cours de ce patient). Et comme d'habitude, le respect du pattern MVC qui fait que chaque vue ne passe que par son contrôleur pour faire quoi que ce soit.

Vue : *(package cabmed.manage.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 vue principale (en fonction de la personne connectée comme déjà dit plus haut).

Vues Secrétaire : (package cabmed.manage.ihm.secretaire)

(1) – La vue *VueRecherchePatient* est celle lancée par défaut lorsque l'on se connecte comme secrétaire. Le nom est assez clair, à partir de cette vue on recherche un patient. Il y a deux façons de retrouver un patient : premièrement, on tape le numéro de registre national du patient

(2) – La vue *VueHistoriquePatient* est

(3) – La vue *VueModifRdv* est

Vues Médecins : (package cabmed.manage.ihm.medecin)

(1) – La vue *VueRechercheDossier* est celle lancée par défaut lorsque l'on se connecte comme médecin.

(2) – La vue *VueDossierPatient* est

(3) – La vue *VueDetailsRdv* est

(4) – La vue *VueAddConclusion* est

f) Application Web

Abc

3.2. Exemples d'interaction :

4. DIVERS :

4.1. Patterns :

Qu'est-ce qu'un pattern ? Un pattern est une façon d'agencer son code de façon à le rendre plus cohérent dans certains cas, ré-utilisable dans d'autres cas, ... C'est vraiment l'organisation du code lui-même qui est visé ici. Quand je parle du code, je parle aussi des fichiers eux-mêmes. Séparer certaines choses dans des fichiers différents c'est indirectement toucher au code.

Dans ce projet j'ai utilisé plusieurs patterns dans différentes situations. Certains qui vont ensemble, d'autres complètement indépendants. Ici je ne parlerais que de quatre patterns (dont deux liés) et laisserai de côté les autres patterns pour ne pas trop alourdir (comme le pattern MVC que j'ai bien évidemment respecter dans le projet).

a) Singleton

Le but du pattern singleton

b) Façade – Observer

c) DAO

4.2. Parties de codes "intéressantes" :

a) Gestion de la navigation entre fenêtres

b) Lecture carte identité

c) Gestion du planning