Niniana RAHITA p1805736
Vladimir KREMNEV p180664

Rapport Projet MIF01 – Mon Espace Santé

Table des matières

Niniana RAHITA p1805736
Vladimir KREMNEV p180664

I) Présentation du projet

L'objectif de ce projet est de réfléchir au développement d'un outil de gestion de données de santé. Notre application a été développée en Java et est inspirée de https://www.monespacesante.fr/. Nous avons mis en place différents principes et méthodes de conception afin de structurer notre code et appliquer les principes fondamentaux de la programmation orientée objet.

II) Environnement de développement

Niniana : Kali Linux, VS Code Studio (1.73.1), OpenJDK 11.0.16

Vladimir : Linux sous WSL 2, VS Code Studio (1.73.1), OpenJDK 11.0.16

III) Design Patterns implémentés

a) Pattern architectural MVC

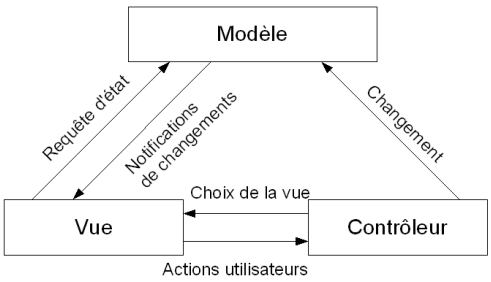

Figure 1- https://baptiste-wicht.developpez.com/tutoriels/conception/mvc/#LII

Modèle (logique métier)

Localisation : /model

La classe MES représente le cœur de l'application, elle est donc équivalente au modèle. Le modèle est responsable de charger et regrouper toutes les données initiales ainsi que de transmettre les demandes d'ajout, modification et suppression des données (indirectement). Les données gérées par MES sont celles des classes Patient, HealthProfessional, Prescription, les différents types de HP, Meeting et Message. Notre modèle ne gère pas seul toutes ces données -> DAO.

Nous avons jugé utile de coupler le modèle avec le DAO afin de ne pas lui affecter trop de responsabilité.

Vue (interface)

Localisation : /view

La classe abstraite View présente les données en cohérence avec l'état du modèle. Elle se charge de capturer et transmettre les actions de l'utilisateur. Nous avons développé six vues qui héritent de View : CreateHPView, CreatePatientView, HomeView, LoginView, ProfilHPView, ProfilPatientView. Ces vues sont initialisées avec des fichiers FXML le langage basé sur XML utilisé par JavaFX. Nous avons également ajouté des dépendances et plugins au pom nécessaires à la visualisation des composants graphiques (javafx-fxml, com.gluonhq).

Certaines fonctions sont directement associées entre FXML<-> Controller et ne passe pas par les classes View, c'est le cas pour tous les boutons déjà configurés par défaut dans le FXML. Cela nous a permis de ne pas alourdir les vues avec des button.setOnAction((event) => { controller.notify() }) qui n'étaient pas nécessaires puisque le controller est directement notifié.

Contrôleur

Localisation : /controller

Le Contrôleur gère les changements d'état du modèle, informe le modèle des actions utilisateur et sélectionne la vue appropriée. Un seul contrôleur pour six vues étant trop lourd (et donnant trop de responsabilités à une classe) nous avons opté pour l'implémentation de six contrôleurs => un contrôleur différent pour chaque cas d'utilisation : Création d'un HP, Création d'un Patient, Connexion, Interaction Profil Patient et Interaction Profil HP, chacun en charge d'une vue et héritant de la classe abstraite Controller.

Il analyse le scénario courant et notifie le modèle de l'évènement associé à un cas d'utilisation.
Nous avons choisi d'implémenter une version dite active du modèle :

- MES change indépendamment du modèle puisqu'il passe par les DAO qui sont inaccessibles au contrôleur
- Une fois les modifications effectuées le modèle informe ses observateurs
- C'est les vues qui prennent cette information en compte
- La vue interroge parfois le modèle pour récupérer l'état de certaine ressource quand il n'y a pas besoin de les modifier (pull)

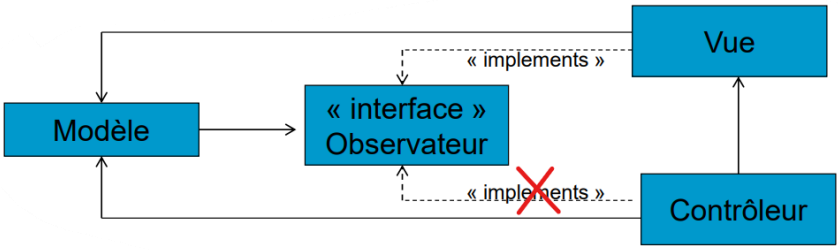

Figure 2 - CM Design Pattern : Lien entre Pattern Observer / MVC

Dans notre cas le contrôleur n'a pas besoin d'être informé puisqu'il sait qu'il y a changement car c'est lui qui délègue au modèle

La vue est appelée en première par l'application (App.java) => initialise en premier le fichier FXML qui lui est rattaché à un contrôleur unique (NomFxmLController.java) => le contrôleur n'a pas encore accès au modèle quand il est appelé par le FXML => une fois le FXML chargé la vue récupère le contrôleur de FXML (fxml. getController()) et le copie avec le bon type (Polymorphisme) => rajoute un lien vers la vue actuelle au contrôleur ainsi que le modèle.

Ce pattern n'était pas un choix de notre part puisqu'il était demandé de l'implémenter dans l'énoncé. Cependant, il a été très bénéfique de l'intégrer à notre structure puisque cette séparation des responsabilités permet une meilleure répartition du travail et facilite la maintenance. Le fait de découper la structure de la sorte à faciliter l'intégration et l'interaction avec les différentes vues et l'ajout d'autres Pattern.

Niniana RAHITA p1805736
Vladimir KREMNEV p180664

b) Pattern comportemental Observer :

Localisation : /observer

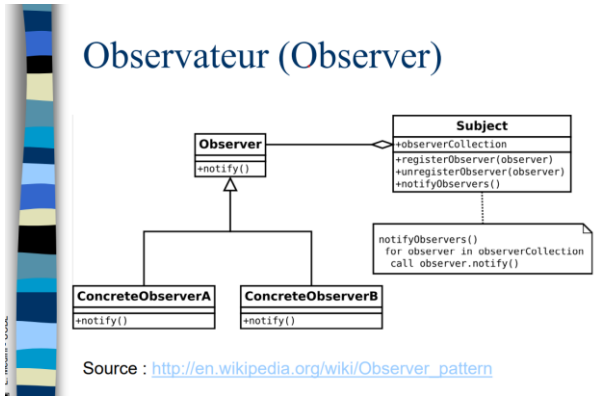

Figure 3- CM Design Pattern

Les observateurs (*Observer)* doivent, comme leur nom l'indique, observer le modèle et réagir en cas de notification. Dans notre structure se sont les vues qui sont les observateurs. Les vues modifient la composante graphique concernée lorsque le modèle les notifie d'un changement.

Le modèle possède une liste d'Observer privée qui lui permet de les notifier. Le contrôleur s'occupe d'ajouter ou supprimer les Observer à la liste du modèle.

Certaines vues n'ont pas besoin d'observer le modèle mais par défaut elles sont ajoutées à la Liste d'Observer du modèle (puisque fonction héritée par View) et elles sont supprimées de la liste d'Observer au premier changement de vue. Nous avons trouvé plus simple de faire hériter cette fonction aux classes Observer afin de nous laisser la possibilité de modifier le comportement de ces vues plus tard.

Nous avons choisi d'intégrer ce pattern avec notre propre implémentation (plus simple et plus léger) : une interface *Observer* implémentée par les vues qui lorsque le modèle les notifie (*notifyObservers()*) lance leur fonction *update()* et une interface *Observable* implémentée par le modèle qui lance la fonction *notifyObservers()*.

c) Pattern créatif Factory

Localisation : /factory

Le pattern Factory a pour but de créer les instances des objets. Contrairement à la création "classique" qui appelle directement les constructeurs de classe, ce pattern permet de séparer les responsabilités de création et de plus facilement gérer la création lorsqu'elle devient trop complexe.

Dans notre projet, nous avons mis en place une Factory pour les patients et les médecins. Pour les médecins, la création n'est pas très complexe et nécessite seulement le type de médecin qu'on veut créer (Généraliste, Dentiste, etc.), mais ce pattern pourra être utile dans le futur, si jamais nous déciderons d'ajouter d'autre caractéristiques aux utilisateurs (Exemple : Patient Politique, Patient VIP).

d) Pattern architectural DAO

Localisation : /daos

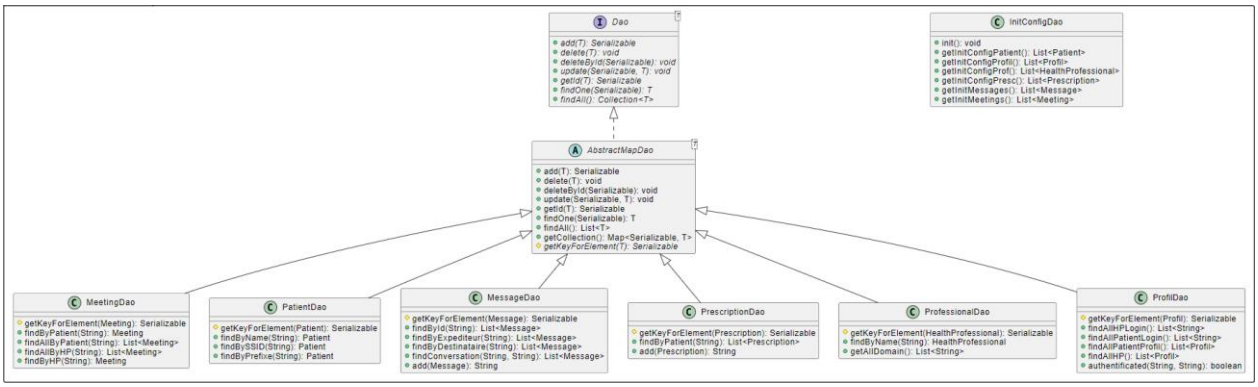

Figure 4- Diagramme de classes DAO PlantUML

Le pattern DAO (Data Access Object) permet de faire le lien entre la couche métier et la couche persistante. Chaque DAO s'occupe de créer, mettre à jour, lire, supprimer des données des Object qu'ils contiennent. Ils héritent de la classe AbstractMapDao qui est une classe-conteneur. Nous avons fait en sorte d'attribuer un objet métier pour chaque DAO.

Pour rendre cette base de données persistantes nous avons créés un fichier initconfig.xml qui est récupéré par une DAO responsable de lire le xml et de renvoyer les listes initialisées aux autres DAO.

Nous avons trouvé intéressant d'intégrer ce pattern notamment pour tester la vue, le fait d'avoir directement des patient et médecins crées fait gagner beaucoup de temps pour le débug mais pas seulement car les DAO permettent d'alléger le modèle. De plus le fait d'implémenter cette base de données facilite une potentielle exportation des classes en vrai base données => création de table (un DAO = une table).

IV) Réflexion éthique

L'objectif principal de Mon Espace Santé est de gérer des documents de santé tels que les traitements suivis, les résultats d'examens, les antécédents médicaux, les compte-rendu d'hospitalisation, qui peuvent être partagés avec les professionnels de santé. L'Assurance Maladie et le ministère de la Santé, avec le soutien du Gouvernement, ont décidé de mettre à disposition cette outil pour simplifier au quotidien les démarches des citoyens mais le déploiement de cette application soulève de nombreux·ses mécontentement/questions notamment au sujet de la sécurité et de la confidentialité des données. On peut trouver de nombreux articles sur internet qui débattent sur ce sujet. Les principales inquiétudes des potentiels utilisateurs :

- Le code du service est géré par une entreprise privée et n'est ni public ni accessible ce qui pose la question de la transparence pour un outil du service public. (Atos/Wordline)
- Comment faire si on ne veut pas de compte ?
- Un cloisonnement des informations insuffisant vis-à-vis du personnel soignant
- Le dispositif d'accès en cas d'urgence ne convient pas à tout le monde.

- Pourquoi ne pas utiliser le chiffrement de données de bout-en-bout comme DoctoLib ?

Mesures, légales et techniques, pour limiter les risques :

- Chacun reste libre de s'opposer à la création de son espace personnel ou de le fermer à tout moment => Droit de suppression et d'opposition
- Mise en place d'une messagerie sécurisée : une adresse de messagerie MSSanté est automatiquement attribuée à la personne usagère et rattachée à Mon Espace Santé. Cette adresse est constituée à partir du matricule INS de l'usagère et du nom de domaine de l'Opérateur de Mon Espace Santé
- Les messages échangés sont stockés pendant une durée de dix ans, sauf lorsqu'ils sont supprimés directement par l'utilisateur·ice.
- Traçabilité des accès à notre profil

Les mesures de sécurité et confidentialité dans notre application :

- Le patient a la possibilité de supprimer ses propres prescriptions ce qui n'est pas le cas dans Mon Espace Santé.
  - 👍 : le patient est responsable de ses données
  - 👎 : une prescription peut être supprimée par erreur
- Pas d'autorisation d'accès par le professionnel de santé
  - 👍 : pas nécessaire si c'est lui qui prescrit
  - 👎 : c'est toujours mieux de laisser le choix à l'utilisateur
- Rendez-vous : seul le professionnel peut en programmer un
- Création d'un espace utilisateur

Les mesures qu'on aurait dues ajoutées dans le cadre d'une vraie application :

- Les fichiers contenant les données devraient être chiffré
- Contrôle d'accès aux infos des patients et médecins
- Double authentification
- Vérification par mail lors de la création
- Système de chiffrage des messages

Nous avons abordé les points critiques au niveau informatique mais MES relève également des enjeux sociaux et environnementaux qu'on ne développera pas en détail dans ce rapport :

- Le cas d'une patiente ayant subi une IVG qui ne souhaite pas divulguer les résultats à ses parents
- Le stockage des données de tous les assurés du système français => base de serveurs => émission de Co2 + consommation d'électricité dont on manque aujourd'hui => conséquences environnementales

V) Tests

A) Tests automatiques

Localisation : /test/java/.../mes/

Les tests unitaires implémentés pour ce projet sont repartis en 3 catégories :

- Les tests pour le modèle et les classes fondamentales (HealthProfessional, Patient, Prescription, etc.)
- Les tests pour les différentes DAO et leur initialisation à partir d'un fichier xml, contenant les différentes patients et médecins, pour avoir un programme avec des données pré rempli
- Les tests pour les contrôleurs, qui vérifie le bon fonctionnement des méthodes n'ayant pas explicitement besoin d'une vue (ajout, suppression, etc.)

Ces tests ont pour objectif de vérifier le bon fonctionnement des méthodes clés comme l'initialisation, la création, l'ajout et suppression, ainsi que le lancement des exceptions pour les fonctions qui les ont.

B) Tests manuels

Nous avons testé manuellement les classes View avec des System.out. print (valueTextField) car nous n'avons pas réussi à accéder aux FXML depuis le dossier où sont lancés les tests et qui sont indispensables à l'initialisation des vues.

C) Tests sur une machine vierge

Environnement : Windows 10, WSL 1 sur distribution Linux Ubuntu, OpenJDK version "11.0.17" 2022-10-18

Tout se lance du premier coup. Un warning apparaît qui n'est pas présent sur nos environnements respectifs :

Loading library prism_es2 from resource failed: java.lang.UnsatisfiedLinkError: ~/.openjfx/cache/19-ea+8/amd64/libprism_es2.so: libXxf86vm.so.1: cannot open shared object file: No such file or directory

java.lang.UnsatisfiedLinkError: libXxf86vm.so.1: cannot open shared object file: No such file or directory

Ce warning n'entrave pas le bon fonctionnement de l'application.

VI) Pistes d'amélioration de notre application

- Séparation du fichier initConfig.xml pour chaque objet métier (Patient, HP, Prescription, etc)
- Rendre persistante les nouvelles données ajoutées à chaque exécution
- Vérification de compte existants
- Suivi de rdv plus poussé
- Couplage du Pattern Factory aux DAO et aux FXML :

Niniana RAHITA p1805736
Vladimir KREMNEV p180664

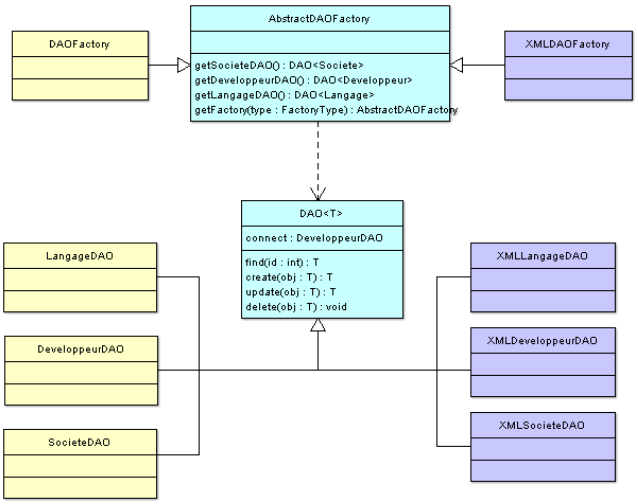

Figure 5- Exemple d'implémentation

VII) Sources

Blog : Pourquoi s'opposer à la création de Mon Espace Santé ? : https://blogs.mediapart.fr/la-quadrature-du-net/blog/280322/pourquoi-s-opposer-la-creation-de-mon-espace-sante

Site officiel de Mon Espace Santé : https://www.ameli.fr/rhone/assure/sante/mon-espace-sante

CM pattern : https://perso.liris.cnrs.fr/lionel.medini/enseignement/M1IF01/CM-patterns.pdf

Cours sur les DAO : https://cyrille-herby.developpez.com/tutoriels/java/mapper-sa-base-donnees-avec-pattern-dao/#:~:text=Le%20pattern%20DAO%20(Data%20Access,stockage%20et%20nos%20objets%20Java.

Thread Twitter : https://twitter.com/GuillaumeRozier/status/1527587327165579266

Icones utilisées dans le projet : https://www.flaticon.com