Application d'Aide Aux Personnes Vulnérables - AAAPV

(lien GitHub : https://github.com/Achraf23/AAAPV)

Introduction

Ce projet avait pour objectif de nous faire découvrir les différentes méthodes et outils de développement logiciel au travers d'un projet.
Pour cela, nous avons développé une Application d'Aide Aux Personnes Vulnérables (AAAPV) centralisée qui permet à des usagers isolés de bénéficier d'aide bénévole. Nous avons ainsi une base de données mise à notre disposition qui nous permet de stocker et de partager les informations entre les utilisateurs.

Cette application permet, dans un premier temps, à un utilisateur demandeur d'aide ou bénévole de créer un compte, ou de se connecter.
Un utilisateur demandeur d'aide pourra poster une mission en indiquant les différentes précisions sur ses besoins et dans son espace personnel, il pourra accéder à ses missions en cours.
Une fois postées, les missions seront visibles par les utilisateurs bénévoles, qui pourront alors choisir de réaliser une ou plusieurs missions.
Par la suite, un personnel médical responsable de l'usager en difficulté pourra valider ou non si le bénévole est apte à réaliser la mission sélectionnée. Dans le cadre de ce projet, en raison du temps imparti, nous avons fait le choix de ne pas développer cette fonctionnalité, ainsi un bénévole qui demande une mission se la voit validée automatiquement.
Cependant, pour faciliter la suite du développement applicatif que nous réaliserons en 5A, nous avons implémenté la possibilité de créer un compte et de se connecter en tant que valideur (sans qu'aucune fonctionnalité ne soit pour l'instant mise à disposition une fois connecté).

Pour nous aider tout au long de notre développement, nous avons utilisé l'outil Jira pour nous répartir les tâches et suivre notre avancée. L'objectif de ce projet étant de nous familiariser avec des méthodes de conception, telles que la méthode agile, nous avons choisi d'implémenter cette application en utilisant le modèle MVC (Modèle Vue Contrôleur).

Après avoir définit les principaux cas d'utilisation, nous nous somme rapidement réparti les tâches comme suit :

- Mise en place de l'automatisation, de la connexion avec la Database (connexion et classe avec méthodes associées) réalisé par Achraf
- Réalisation de la Vue (GUI) par Océane
- Implémentation du modèle et des différents contrôleurs (lien entre vue et modèle) par Emilie

1. Implémentation du modèle et du contrôleur

Lors de notre implémentation du modèle, nous avons choisi d'écrire 3 classes héritant de la classe User, qui est générale et permet d'utiliser le polymorphisme. Les trois classes "*Vulnerable*" (pour les demandeurs d'aide), "*Volunteer*" (pour les bénévoles) et "*Validator*" (pour le personnel médical) redéfinissent la méthode equals avec l'ajout de la comparaison de l'attribut "type" entre deux objets. Cet attribut "type" est un *Enum* qui permet de différencier les trois types d'utilisateurs.

Le fait d'avoir une classe mère User permet dans le controller de vérifier la présence ou non d'un utilisateur (tout type confondu) dans la BDD au moment de la connexion.
Cela est géré par la classe abstraite "*ControllerUser*", avec la méthode "*addConnexionListener*" qui est appelée dans la vue (classe GUI) lorsque l'utilisateur a entré ses informations personnelles et confirme la connexion en appuyant sur un bouton.

De façon générale, les différentes classes controller permettent de gérer les actionListener, qui sont les méthodes appelées lors d'un clic sur un bouton par exemple. Ainsi, au moment de la phase de connexion, si l'utilisateur est déjà présent dans la BDD, et que le mot de passe entré est correct alors la connexion se fait, sinon, si l'adresse mail est déjà utilisée, l'utilisateur est invité à entrer le bon mot de passe. Si aucun de ces deux cas n'est remplit, alors un nouvel utilisateur est créé et ajouté à la BDD.

Pour finir, comme la classe "*ControllerUser*" est abstraite, elle définit des méthodes abstraites redéfinies dans les trois autres controller, adapté aux actions possibles pour les différents types d'utilisateurs. Par exemple, la méthode "*homepage()*" pour les utilisateurs vulnérables lance une nouvelle fenêtre avec la possibilité de créer et poster une nouvelle mission, ou d'afficher ses missions en cours.

Afin de faciliter les interactions avec la database, nous avons décidé d'écrire des méthodes statiques et publiques dans la classe Database. Pour permettre la connexion, il a bien sûr fallu rajouter une dépendance MySQL dans le *pom.xml*. Comme nous avons deux tables dans la database, nous avons implémenté des méthodes pour ajouter une ligne dans une des deux tables ou bien effacer toutes les lignes (pour réinitialiser les tables avant le lancement de l'application dans les tests).

Nous avons utilisé maven pour développer notre application afin de mettre en place une intégration continue. Nous avons également mis en place une automation du développement avec Github grâce à un workflow lancé à chaque push de code d'un collaborateur.

2. Implémentation de la vue

L'implémentation de la vue (qui se trouve dans la classe "*GUI*") a été faite avec Swing. L'application s'utilise "en cascade", avec des actions qui ouvrent d'autres fenêtres (*JFrame*) permettant de faire d'autres actions qui ouvriront d'autres *JFrame*, *et cetera*. Par exemple, la première page d'accueil permet à l'utilisateur•trice de "définir" son type (soit Vulnérable, Bénévole ou Valideur) via des *JButtons*, et chaque bouton emmène vers le formulaire de connexion/inscription.
Le formulaire contient des *JTextField* qui récupèrent les valeurs entrées par l'utilisateur (grâce au contrôleur) – on notera que le champ password est un *JPasswordField*, qui permet de masquer les caractères utilisés par des astérisques. Le bouton "*Connexion"* permet ensuite d'accéder à une nouvelle *JFrame*, correspondant à la page d'accueil personnelle de l'utilisateur.

Selon le type de l'utilisateur, sa page d'accueil personnelle contient différentes informations :

- Pour les demandeur•euses d'aide, on y trouve deux boutons : "*Demander de l'aide*", qui emmène vers un formulaire de demande d'aide, et un bouton "*Gérer vos demandes en cours*", qui emmène vers une fenêtre affichant la liste des missions de l'utilisateur•trice en cours de traitement.
- Pour les bénévoles, on y trouve la liste des demandes d'aides qui n'ont pas encore de bénévoles, ainsi qu'un bouton "*Accepter la demande*" qui place le•a bénévole sur la mission sélectionnée sur la page d'accueil, ainsi qu'un bouton "*Gérer vos demandes en cours*" (idem que pour les demandeur•euses).
- Pour les valideur•euses, on y trouve la liste des demandes d'aides qui attendent une validation (donc qui ont déjà un•e bénévole de placé•e), ainsi qu'un bouton "*Accepter une demande*" et "*Refuser une demande*" — boutons qui ne sont pas fonctionnels par manque de temps pour leur implémentation.

3. Implémentation des tests

Pour finir, nous avons réalisé des tests automatisés avec Junit en ajoutant les dépendances nécessaires dans le fichier pom.xml.
La classe "*UserTest*" permet de tester si la création de différents utilisateurs en fonction de leur type est bien réalisée et ensuite de vérifier l'implémentation de la méthode equals par exemple.
La classe "*DatabaseTest*" s'assure du bon fonctionnement de la connexion à la database ainsi que l'envoi des requêtes. Nous avons mis en place des tests basiques : vérifier que la requête addUser fonctionne bien, vérifier que *getAllMissions()* nous rend bien une unique mission après avoir ajouté une seule mission dans la table.