CV Search

Un essai d'outil de gestion de CV assisté par ordinateur

Fourkane Saïd Ali
Christophe Marrel

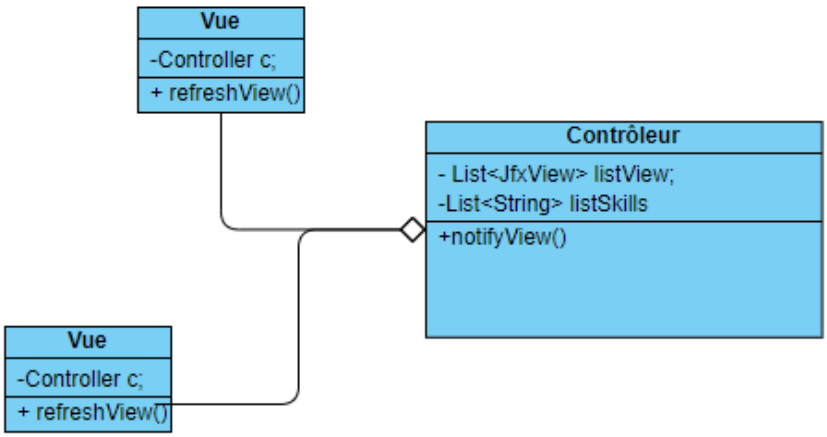
7 Novembre 2021

Introduction

Notre application permet à un recruteur de sélectionner des candidats parmi une liste de candidats. Chaque candidat possède une liste de compétences avec une note comprise entre 0 et 100 par compétence permettant d'évaluer son niveau ainsi que ses expériences professionnelles. Chaque expérience possède une durée et des mots clés. Le recruteur a la possibilité de choisir les compétences qu'il recherche ainsi qu'une stratégie de sélection parmi plusieurs permettant de filtrer le candidat selon son niveau ou ses expériences. Le recruteur peut ajouter des mots clés en les renseignant un par un dans la barre de recherche et en appuyant sur le bouton « Add skill ». Il peut également en supprimer un en cliquant sur la croix à côté de la compétence. Il a ensuite un sélecteur lui permettant de choisir une stratégie de sélection parmi plusieurs. Enfin, lorsqu'il appuie sur le bouton « search », la recherche est enregistrée et il obtient la liste des candidats correspondant à cette recherche avec leur nom et leur moyenne dans les compétences sélectionnées (ou la durée moyenne en année d'expérience en cas de stratégie sur l'année d'expérience). Les candidats sont triés selon leur moyenne décroissante de la stratégie.

Nous avons également rajouté un bouton « Restore » permettant au recruteur de restaurer l'état précédant de sa recherche dans le cas où il a effectué plusieurs recherches.

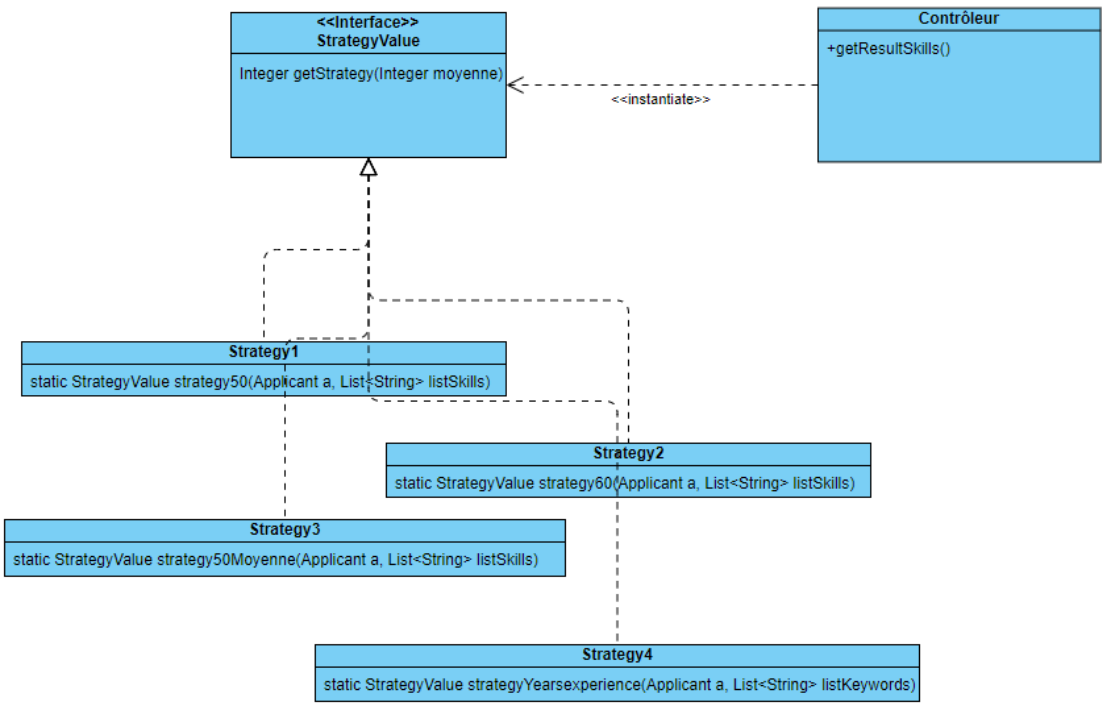
Design patterns



Observer :

Le patron observateur est un patron de conception qui est utilisé pour envoyer un signal à des modules qui jouent le rôle d'observateurs. En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui lui parviennent depuis les modules qu'ils observent (les observables).

Dans ce projet, la vue va modifier la liste contenant les skills du contrôleur. Une fois modifiée, elle va appeler grâce à la méthode notifyView() contenue dans le contrôleur, toutes les méthodes contenues dans les autres vues (y compris la sienne) de refreshView(). Cette méthode va lire la nouvelle liste de Skills, et ainsi changer la vue en fonction du nouveau contenu.



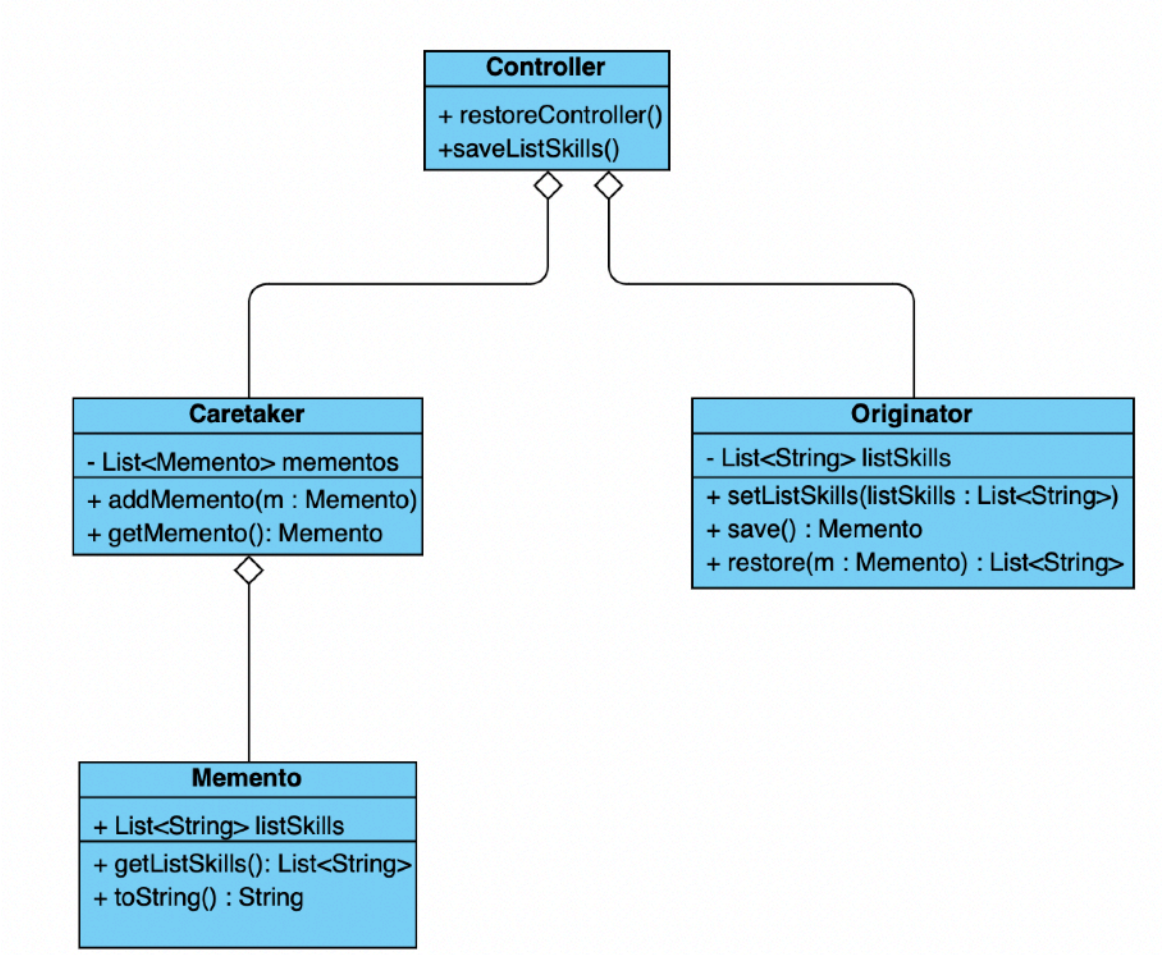
Strategy :

Le contrôleur va créer les instances de l'interface StrategyValue avec la stratégie correspondante à celle choisie par l'utilisateur, cette stratégie va au

final renvoyer la valeur demandée au contrôleur qui va ainsi pouvoir la fournir à la vue.

Le contrôleur contient une Hashmap de toutes les instances des stratégies. La clé de cette map est la donnée contenue dans le modèle de Strategy. Ainsi, grâce au choix de l'utilisateur, le contrôleur appelle directement la bonne stratégie et renvoie la valeur attendue en fonction de l'Applicant et de la liste des Skills.

Memento :



Nous avons décidé de nous servir du pattern de comportement Memento afin d'intégrer une fonctionnalité permettant de restaurer un état précédant du champ de Skills d'un recruteur via le bouton « restore ».

Pour cela, nous avons rajouté trois classes dans notre package contrôleur. La classe Memento contient un état à un instant T d'une liste de compétences. Il contient la liste de compétences qu'un recruteur a recherché à un moment.

Ensuite, la classe Caretaker possède une liste des mementos. Elle permet d'enregistrer l'ensemble des états recherchés. Enfin, la classe Originator possède une donnée membre Memento ainsi que deux méthodes : save et restore. La fonction save permet de retourner le memento actuel, et la fonction restore prends en paramètre un mémento et retourne sa liste de compétences.

Le contrôleur possède une instance Caretaker, une instance Originator ainsi que des procédures saveListSkills et restoreController.

La procédure saveListSkills permet d'ajouter la liste des compétences recherchées dans le Caretaker du contrôleur. Elle est appelée dans la vue, à chaque fois que le recruteur effectue une recherche.

La procédure restoreController restaure la liste des compétences du contrôleur via la liste des enregistrements de Caretaker.

Éthique

L'algorithme peut être mis en défaut dans des cas où un humain aurait fait un meilleur travail si on prends en compte uniquement l'expérience sans prendre en compte son niveau. On aurait donc des informations limitées sur le candidat. Par exemple, on pourrait rater un candidat doué mais peu expérimenté.

Si le recruteur préfère choisir un candidat selon son niveau, il peut toujours choisir une stratégie opérant selon le niveau, mais dans ce cas, il risque aussi de rater quelqu'un d'expérimenté. Afin de palier à ce problème, il faudrait rajouter des stratégies permettant à la fois de traiter l'expérience d'un candidat et son niveau.

L'algorithme en soit n'est pas discriminatoire car il n'a pas accès à d'autres informations que les compétences et le niveau (et non accès a d'autres informations comme le sexe, l'âge, lieu de naissance/ville actuelle, diplôme) et il ne fait aucun traitement sur les noms/prénoms.

Tests

Nous avons réalisé différents tests au niveau des modèles et du contrôleur.

Dans la classe ApplicantTest, les tests permettaient de vérifier qu'il est possible de récupérer un candidat depuis un fichier applicant.yaml. Nous avons rajouté 3 tests correspondant à l'extension de la classe Applicant : la prise en compte de l'expérience. Le test testIfInKeyword() vérifie que la fonction membre inkeywords() de la classe Applicant retourne vrai seulement si le

paramètre fait parti des mots clés du candidat. Nous avons également réalisé le test testGetYearsOfExperience() permettant de vérifier qu'elle retourne bien le nombre d'année d'expérience d'un candidat selon un mot clé. Enfin, le test testListOfKeywords() permet de vérifier que la fonction listOkKeywords récupère bien la liste de tous les mots clés d'un candidat.

La classe ControlerTest contient un test sur le contrôleur. Il permet de tester différentes stratégies avec comme recherche « c++ » et vérifie que lorsqu'on applique chacune des stratégies, on récupère bien les bons candidats avec la bonne moyenne.

La classe ExperienceTest permet de tester qu'on récupère bien la liste des expériences d'un candidat avec l'ApplicantListBuilder.

Le test MementoTest permet de vérifier la fonctionnalité memento. Dans ce test, on enregistre différents états d'une liste de skills, correspondant aux compétences recherchés par le recruteur. On vérifie ensuite que lorsqu'on appelle la fonction Restore, la dernière recherche est bien supprimée.

La classe ModelTest contient un test permettant de tester différentes stratégies. Ce test permet de vérifier que lorsqu'on instance une stratégie, on obtient bien les bons résultats.

Pour le test de la vue et du modèle de conception Observer, nous avons fait le test à la main. Pour la vue, lorsque nous appuyons sur un bouton cela doit bien afficher ce que le bouton doit afficher, par exemple lorsque nous ajoutons un Skill il doit être affiché en dessous avec un bouton permettant de le supprimer. Et lorsque nous avons choisit une stratégie, l'application doit nous afficher (après appui sur le bouton search) le résultat de cette stratégie.

Pour le test du modèle de conception de l'Observer, une fois un skill ajouté les autres instances de vue doivent afficher ce nouveau skill. Et lorsque nous supprimons ce skill d'une vue, toutes les instances se mettent à jour.

Pour la création de l'UML nous avons utilisé le plugin Maven com.iluwatar.urm. Qui au début nous donnait un fichier dot, que nous avons transformé en png.