Rapport du Mini Projet « CV Search »

Projet de recherche de participants par C.V.

Mohamed El Azzouzi et Noauffal Abdullatief

06/11/2021

MIF01 – Master 1

INTRODUCTION

Java est un langage orienté objet, mais relativement confortable à utiliser du fait de sa gestion de la mémoire, permettant la conception de grands projets tel que celui que nous allons présenter ici.

Le projet consiste en la création d'un logiciel de gestion de tri de C.V. par recherche intelligente. L'accent est mis sur une modularité des différentes stratégies mise en place pour effectuer ces recherches.

Nous nous attarderons dans ce document sur la conception et la programmation du logiciel, nous allons ainsi voir premièrement nos choix de conception et les patrons (« *design patterns* ») utilisés et leurs implémentation et justifications, puis nous aborderons le domaine de l'éthique avec quelques explications sur les stratégies et leurs controverses, et enfin il sera sujet des différents tests qui permettent de vérifier la robustesse du code de l'application.

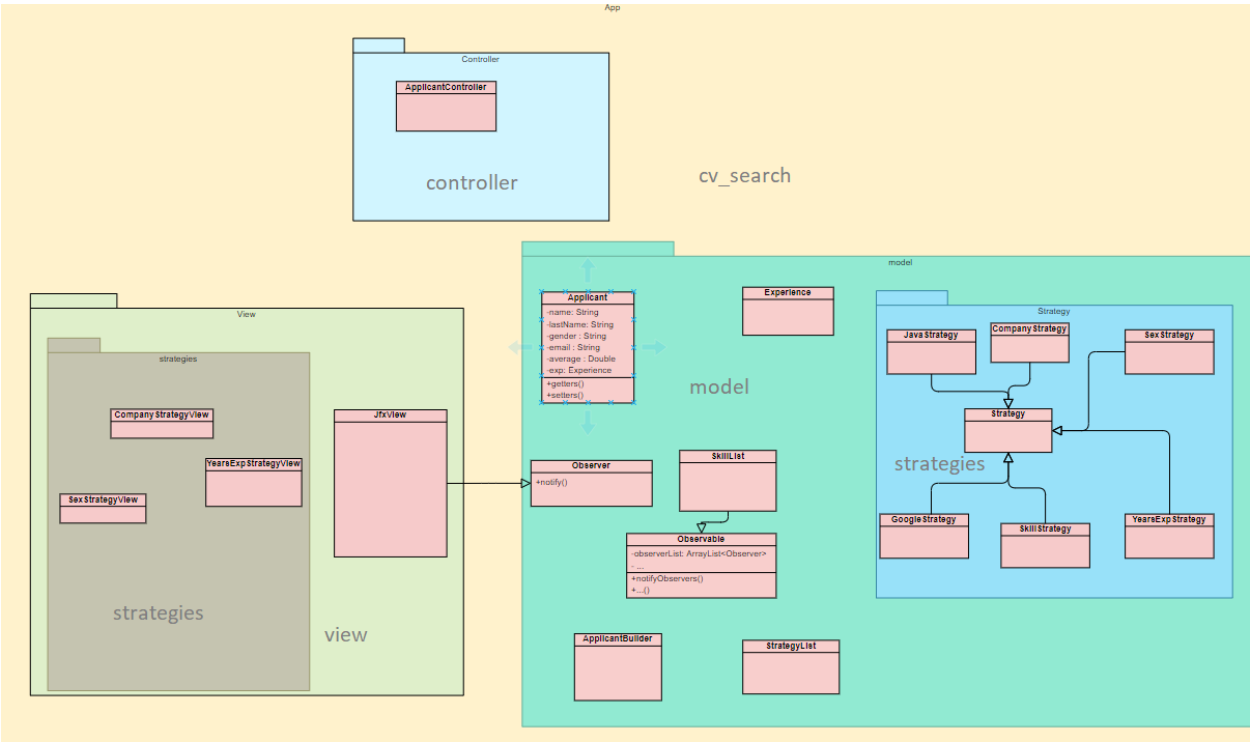
Les schémas et diagrammes montrés ne seront pas complets et serviront simplement à appuyer la lecture.

Conception

La conception s'est orchestrée autour de l'idée d'une modularité des stratégies. Nous avons donc un logiciel avec pour particularité de venir directement regarder les stratégies disponibles (les classes présentes dans l'exécutable) et d'ensuite les afficher et les associer à une vue du même nom si elle existe pour pouvoir paramétrer. Les différents patrons de conceptions utilisés seront détaillés ci-dessous.

Patron modèle-vue-contrôleur.

Très simple : il sert à séparer les responsabilités du programme en 3 catégories. Dans notre projet nous avons un package java pour chacune d'elles. *ApplicantController* est la classe qui contrôle, *JfxView* ainsi que les classes *…View* des stratégies sont des vues et enfin autres classes sont des modèles. Ci-dessous une vue globale du projet et de ces 3 groupes.

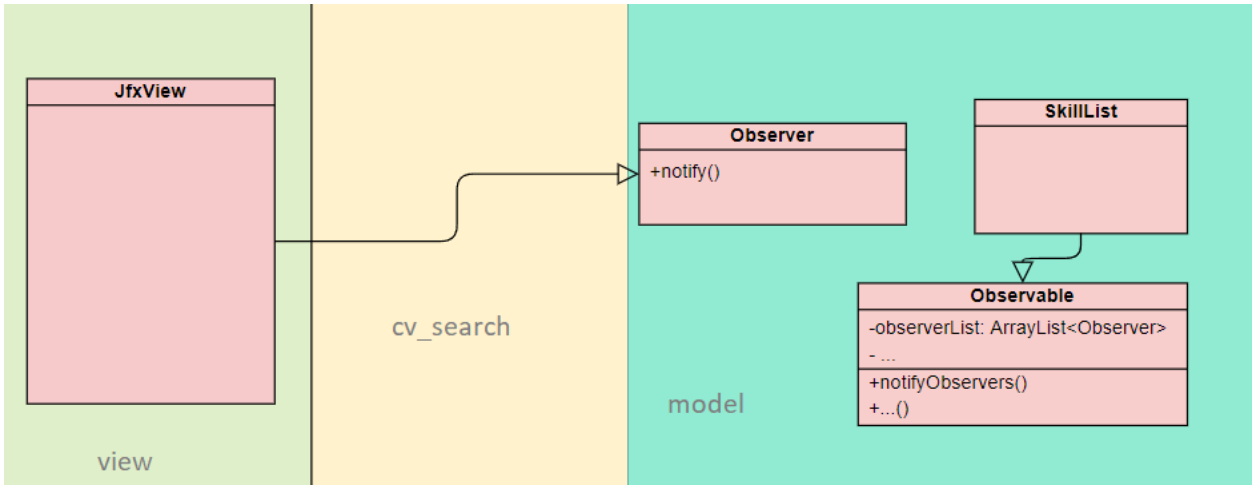


Patron *observer*.

Ce *design* sert de moyen de communication et d'échange d'informations entre des instances de classes de n'importe quel type. Il est particulièrement utile pour des projets de taille moyenne à grande et surtout vient soulager le polymorphisme. Nous l'avons implémenté pour plusieurs utilisations dans le logiciel.
Deux classes, une observatrice et l'autre observable, servent de base et sont étendues aux classes qui ont besoin de ce *pattern*.

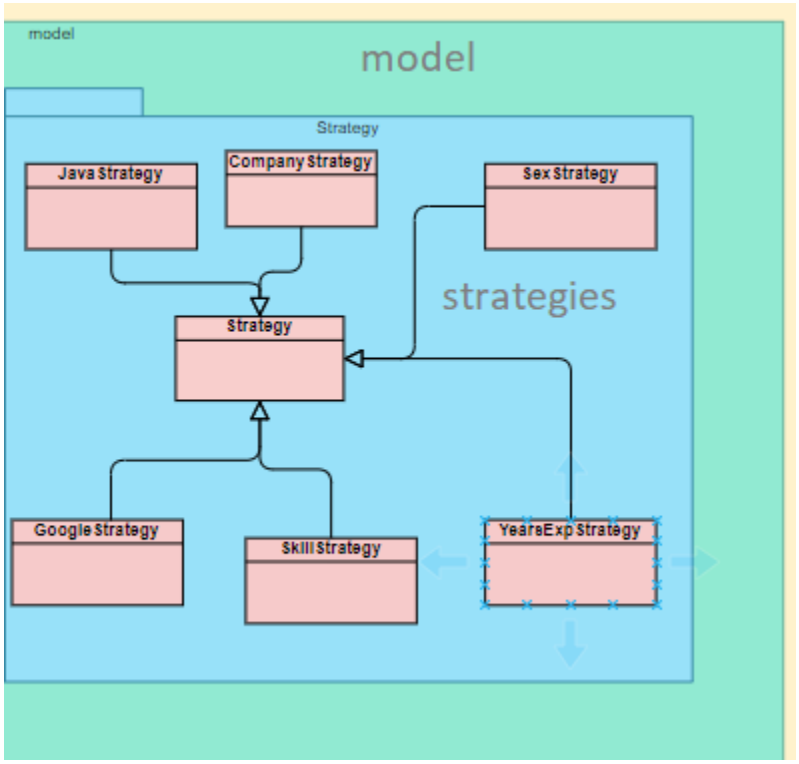
La classe observatrice contient une fonction de notification qui sera appelée par les classes observables qu'elle observe. Ce sont ces dernières – les classes observables – qui s'occupent de garder une liste de leurs observateurs et les notifient lorsqu'il y en a besoin.
Les classes *StrategyList* et *SkillList* (servant respectivement de liste de stratégies et de liste de skills) sont des observables et viennent notifier leurs vues (*JfxView*, des observatrices) lorsqu'elles sont mises à jour (ajout, suppression, etc.) Et les classes dérivées de *Strategy* sont des observatrices qui sont notifiées par leurs vues lors du remplissage de formulaires et après validation.

Voici un schéma de ce patron dans notre code.



Polymorphisme.

Ce *design* est évident, et est indispensable dans un projet orienté objet. Voici un schéma illustrant son utilisation dans notre projet.



Décorateur.

Le schéma de la page précédente illustre également bien ce patron de conception. Ici nos stratégies peuvent être implémentées de manière indépendante sans toucher au reste du code. On peut étendre et ajouter des extensions sans avoir à modifier le code existant.

Il y a d'autres patrons de conceptions (*expert, creator, controller, builder*…) mais nous nous sommes concentrés seulement sur les plus pertinents.

Éthique

Pour cette partie, nous avons opté pour un libéralisme mais en restant sécurisé. En effet, nous avons décidé d'inclure des stratégies de base : la recherche par compétences, par nombre d'années d'experience, mais nous avons également inclus une recherche par sexe, qui – *a prioris* – peut permettre une discrimination, mais qui en réalité sert seulement à respecter les quotas des entreprises car certaines lois en imposent (*reverse discrimination*), par exemple pour avoir plus de femmes. Pour le sexe – dans un soucis de progressisme – nous acceptons tout. Nous n'implémentons pas de recherche par race ou par nom ou prénom car ici il s'agirait sans doute d'encourager les discriminations raciales. Malgré tout cela, les entreprises utilisant ce logiciel peuvent tout à fait implémenter de nouvelles stratégies simplement, et donc potentiellement agir en contraire à nos convictions et à la loi.

Tests

Pour commencer, nous avons réparti les tests sur plusieurs fichiers. Nous avons le fichier *ApplicantTest.java* où sont effectués les tests du fichier *Applicant.java* ainsi que les tests du fichier *Experience.java*. Nous avons les tests de SkillList.java dans le fichier *SkillListTest.java*. Dans le fichier *StrategyTest.java*, nous avons les tests de toutes les stratégies implémentées. Et pour finir nous avons les tests de la *StrategyList.java* dans *StrategyListTest.java*. Ces tests sont effectués sur des fichiers yaml placés dans le dossier *ApplicantTestFiles* pour éviter que le *bot* écrase les fichiers de tests.

Dans les tests d'un candidat (*applicant*), nous testons s'il a été correctement construit par le *builder*. On teste si tous les membres privés sont corrects. Ensuite nous testons l'expérience d'un candidat, on vérifie que l'expérience a correctement été traduite en java. Elle est stockée dans une *Map*, on effectue donc des tests à l'aide de la méthode *get* de *Map/List*.

Pour la « *skill list* », on teste les méthodes *add, get, clear* et *remove.*

Pour la « *strategy list* », on effectue les mêmes tests que pour la *skill list*.

Enfin, pour les stratégies, on teste pour chacune des stratégies que le résultat attendu soit le bon, chaque stratégie est appelée sur un candidat généré à partir d'un fichier test (*ApplicantTestFiles/test1.yaml*).

CONCLUSION

Notre logiciel est complet, il permet la recherche sans erreur de candidats dans une liste de fichiers. Il permet l'utilisation de plusieurs stratégies dynamiques de recherche, stratégies qui sont amenées à évoluer par modifications et extensions. Le logiciel est facilement améliorable car il profite pleinement des possibilités du langage java et de l'orienté objet et ses avantages. Elles filtrent une par une les candidats quand elles sont choisies et paramétrées.

Pour rendre l'application plus réaliste et pour pouvoir tester son bon fonctionnement, nous avons mis en place un programme *bot* écrit en python (prenant en paramètre le nombre d'applications souhaitée) qui permet de générer des candidats (fichier *.yaml*). Nous avons donc pu tester l'application avec plus de 500 candidats. (Ce programme est dans le dépôt *git*.)