Gestion de Projet et Génie Logiciel M1
Université Claude Bernard Lyon1
2025-2026

Rapport Projet “CV-SEARCH”

Otmane El Mouki p2208286
Nedim Trad p2006707

Tables des Matières

Introduction

Dans ce projet, notre objectif n'était pas seulement de créer un outil capable de sélectionner des CV, mais surtout de construire une application propre et facile à faire évoluer. On voulait un code clair, organisé, que n'importe quel développeur puisse comprendre rapidement, sans dépendances inutiles ni logique dispersée.

Plutôt que d'empiler des fonctionnalités, on s'est concentrés sur une architecture solide : un MVC bien séparé, des patterns utilisés pour de bonnes raisons, et des composants qui s'articulent naturellement entre eux. L'idée était de pouvoir ajouter une stratégie de sélection, une nouvelle façon d'afficher les candidats ou une règle métier supplémentaire sans avoir à modifier ce qui existe déjà.

L'outil aide un recruteur à trouver les bons profils, mais c'est surtout un projet pensé pour durer : lisible, extensible et agréable à maintenir.

Design patterns

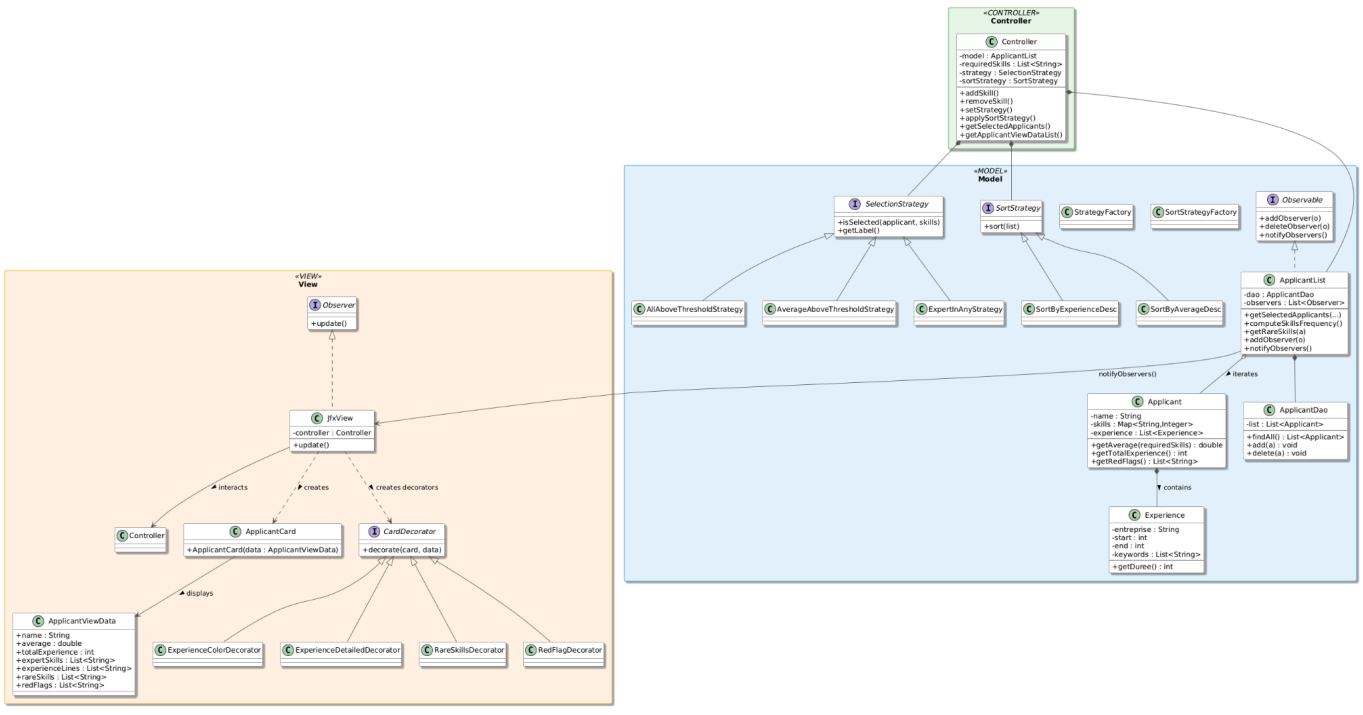


Diagramme UML

MVC : Architecture générale du projet

Lors de nos premières réflexions, la vue accédait directement au modèle : elle récupérait les ***Applicant***, lisait leurs expériences, leurs compétences… C'était simple, mais rapidement nous avons vu le problème : la vue dépendait entièrement de la structure interne du modèle. Le moindre changement dans les classes métier risquait de casser l'affichage.

Ceci nous a poussés à adopter un **MVC strict**. Dans l'architecture finale, la vue ne manipule plus jamais les objets du modèle. Le contrôleur devient l'unique intermédiaire : il interroge le modèle, prépare les données puis les expose à la vue sous forme ***d'ApplicantViewData***, un DTO qui contient uniquement ce qui doit être affiché. La vue reste donc légère et totalement indépendante de la logique métier.

Cette refonte a rendu le code nettement plus robuste : le modèle peut évoluer sans impact sur l'interface, la vue se concentre uniquement sur la présentation, et toute la transformation des données est centralisée là où elle doit l'être, c'est-à-dire dans le contrôleur.

Aujourd'hui, le seul lien qui reste entre modèle et vue passe par le pattern **Observer** : le modèle signale qu'il a changé, la vue réagit en appelant le contrôleur pour récupérer des données fraîches. Le diagramme UML illustre bien ce point : aucune relation directe modèle → vue en dehors de cette notification. Cela assure un découplage maximal et un MVC plutôt bien maîtrisé.

Pattern Observer: synchroniser la vue et le modèle

Nous avons adopté le pattern **Observer** pour garantir une synchronisation propre entre le modèle et l'interface, sans créer aucune dépendance directe. La vue s'enregistre simplement comme observateur du modèle : elle se met donc automatiquement à jour lorsque les données changent mais le modèle n'a absolument aucune connaissance de la structure de la vue ou comment celle-ci fonctionne.

Ce mécanisme présente un avantage important : il fonctionne de la même façon qu'il y ait une seule vue ou plusieurs. Si demain une seconde interface graphique, une console, ou même une API observaient le même modèle, elles recevraient toutes les mêmes notifications et resteraient parfaitement synchronisées. Le modèle ne fait que prévenir : « quelque chose a changé » ; les vues réagissent chacune à leur manière (qui peut être différente aussi selon la manière dont on veut afficher les données).

Dans notre cas, dès que la vue reçoit une notification, elle déclenche `update()`. Cette méthode est volontairement simple : elle appelle trois fonctions distinctes : `refreshSkills()`, `refreshResults()`, `refreshStrategy()` — chacune dédiée à une seule responsabilité (par

exemple refreshResults() s'occupe de l'affichage de la liste des résultats, dans ce cas des candidats). Cette organisation améliore la lisibilité, évite la logique dispersée et respecte les bonnes pratiques de conception.

Pattern Dao : isoler la source des données et préparer l'avenir

Dans notre application, la classe *ApplicantList* ne stocke pas directement la liste des candidats : elle délègue entièrement cette responsabilité à un *ApplicantDao*. Cette décision peut sembler anodine dans la version actuelle où les données sont simplement en mémoire, mais elle a un intérêt structurel.

ApplicantDao devient le point unique où est stockée et manipulée la liste des candidats. Toutes les opérations de gestion , comme l'ajout, suppression, effacement complet, récupération d'éléments passent exclusivement par lui. *ApplicantList*, de son côté se concentre uniquement sur la logique métier, comme la sélection selon une stratégie, le calcul de rareté des compétences ou la mécanique d'observation.

Nous avons donc volontairement évité de placer directement la liste des candidats dans *ApplicantList*. Cela aurait figé l'architecture et empêché toute évolution sans réécrire une partie importante du modèle. Avec le Dao, *ApplicantList* itère simplement sur le résultat de dao.findAll(), ce qui permet à la logique métier de rester totalement indépendante du support sous-jacent.

Dans cette démarche nous avons donc une centralisation claire des données, car le modèle ne traite pas directement la structure de stockage, ce qui rend le code plus propre et plus extensible , par exemple pour ajouter des nouvelles méthodes de traitement de la liste de candidats , on sait exactement où on doit se placer et éviter de casser des composantes importantes.

Un dernier avantage important de l'utilisation de ce Dao, c'est que si demain notre application evolue dans ses besoins, c'est-à-dire qu'il faut mettre une place une base de données pour stocker la liste des applicants , il suffira donc de remplacer l'implémentation du *Dao.ApplicantList* et tout le reste du modèle n'aura pas besoin de changer.

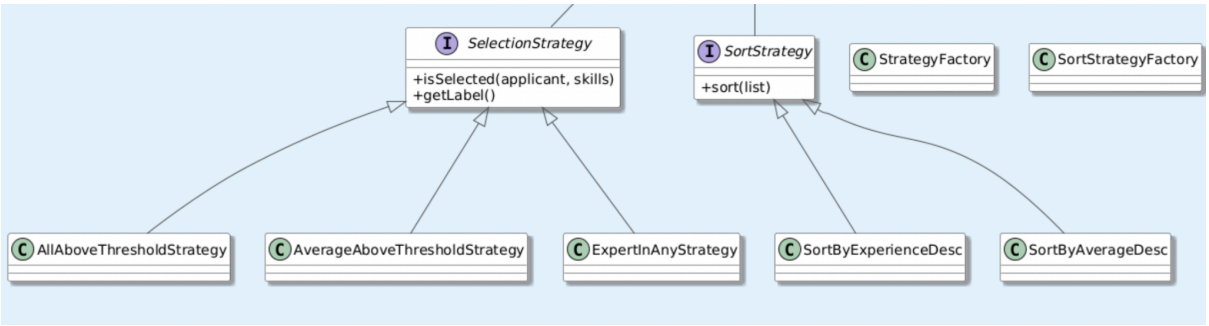
Pattern Factory : un moyen propre d’instancier nos stratégies

Dans notre application, plusieurs stratégies cohabitent : différentes manières de sélectionner les candidats, différentes façons de les trier. Sans organisation particulière, le contrôleur aurait dû connaître chacune de ces classes concrètes, les instancier lui-même, et évoluer à chaque fois qu’une nouvelle stratégie apparaissait. C’est précisément ce que nous voulions éviter : un contrôleur dépendant de tous les détails de création, et donc difficile à maintenir. Nous avons donc introduit deux **factory** dédiées : une pour les stratégies de sélection et une pour les stratégies de tri , qui centralisent entièrement la création de ces objets. Le contrôleur ne manipule plus que les **interfaces abstraites** et ne demande qu’un type logique, sans jamais savoir quelle classe concrète se cache derrière. Cela nous permet d’ajouter de nouvelles stratégies à volonté, sans modifier ni le contrôleur, ni la vue, ni le reste du modèle.

L’intérêt est double : le code reste stable même lorsque de nouveaux comportements apparaissent, et l’architecture gagne en extensibilité. Le diagramme UML met d’ailleurs bien en évidence cette séparation : toutes les stratégies concrètes sont regroupées sous leurs interfaces, et les *factories* sont les seuls points chargés de leur construction.

En pratique, l’usage de ces *factories* se ressent surtout au niveau du contrôleur. Lorsqu’un utilisateur change la stratégie via l’interface (par exemple passer d’une sélection “ALL > un seuil” à “Expert dans au moins une compétence”), le contrôleur ne fait qu’invoquer la *factory* avec le type voulu. Toute la logique de décision, d’instanciation et d’évolution se trouve centralisée derrière la méthode de création.

Ce mécanisme nous a permis d’ajouter de nouvelles stratégies pendant le développement (comme les variantes de seuil ou la stratégie expert, ou autre types de tri) sans toucher une seule ligne dans le contrôleur. C’est donc ce qu’on attend exactement du bon usage de ce pattern , c’est une architecture qui accuille de nouvelles fonctionnalités ou comportements sans augmenter la complexité globale du système.



*Stratégies de sélection et de tri, avec leurs implémentations et les **factories** qui créent la bonne stratégie à la demande.*

Pattern Strategy : rendre la sélection et le tri totalement configurables

Au moment de concevoir la logique de sélection des candidats, on s'est très vite retrouvés face à un problème : selon les recruteurs, les critères peuvent varier énormément. Certains veulent uniquement les candidats maîtrisant toutes les compétences demandées, d'autres préfèrent favoriser les profils ayant une forte expertise dans une skill spécifique, d'autres encore privilégient la moyenne globale des skills. Ajouter toutes ces règles directement dans le modèle aurait rapidement transformé l'application en un ensemble de conditions difficiles à lire, à maintenir et surtout impossible à faire évoluer proprement.

Le pattern **Strategy** s'est imposé donc comme une réponse cohérente à ce problème. Nous avons défini une interface commune (***SelectionStrategy***), et chaque règle de sélection a été encapsulée dans sa propre classe : ***AllAboveThresholdStrategy***, **AverageAboveThresholdStrategy** et **ExpertInAnyStrategy.**
Le modèle n'a plus besoin de connaître les règles ni de savoir « comment » sélectionner un candidat : il reçoit simplement une stratégie et l'applique. Le contrôleur, lui, choisit dynamiquement la stratégie en fonction des actions de l'utilisateur , puis passe cette stratégie au modèle sans jamais toucher à son code interne.

Pour simplifier encore l'intégration, nous avons introduit une **StrategyFactory**, qui centralise la création des stratégies et évite d'éparpiller des new partout dans le code. Cela rend le système plus propre, mais surtout plus extensible : ajouter une stratégie se résume à créer une nouvelle classe et à l'enregistrer dans la factory.

Nous avons appliqué exactement le même principe au tri des candidats. Le tri repose sur une seconde interface (***SortStrategy***), implémentée par ***SortByExperienceDesc*** et **SortByAverageDesc**. Là encore, le contrôleur ne contient aucune logique de tri : il se contente d'appeler la stratégie active et d'afficher le résultat. Passer d'un tri par expérience à un tri par moyenne devient donc un simple changement de stratégie, sans conditionnelle et sans duplication de code. La **SortStrategyFactory** garantit donc la même souplesse que pour la sélection.

Pattern Décorateur : une extension de l'affichage simple et complètement modulable

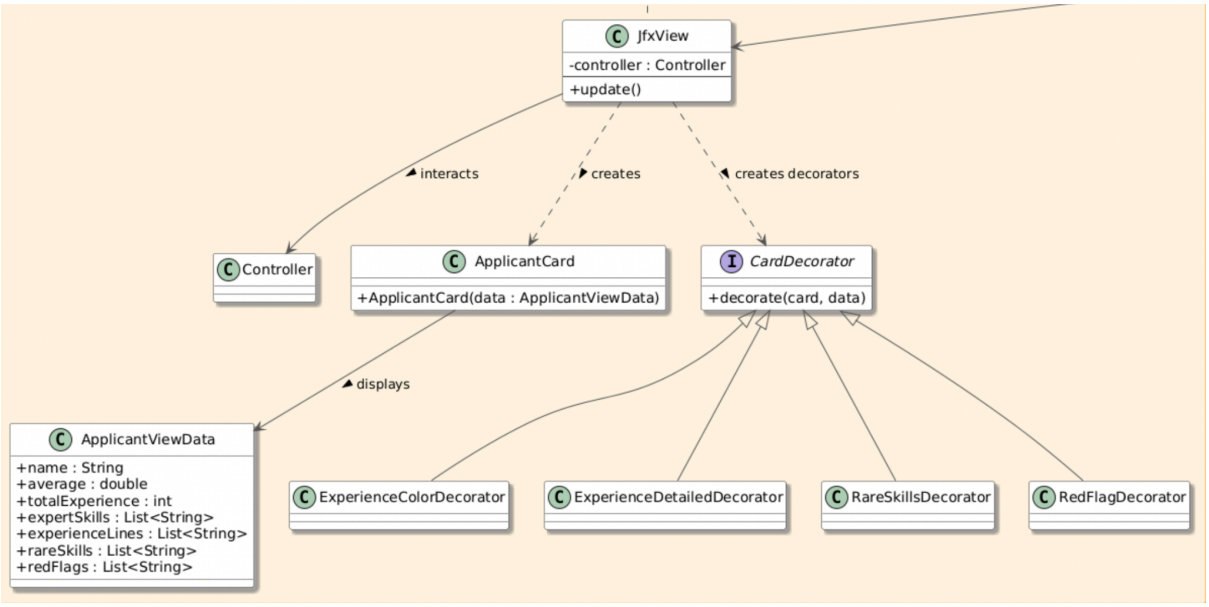
Le **décorateur** est devenu utile au moment où l'interface a commencé à proposer plusieurs options d'affichage : mettre les **cards** des candidats en couleur selon leur expérience, afficher les détails de leurs expériences (ce qu'ils ont réellement fait dans chaque entreprise), mettre en valeur les compétences rares ou encore signaler des red flags. Sans ce pattern, tout aurait fini directement dans *ApplicantCard*, ce qui aurait rapidement transformé cette classe en un bloc lourd, difficile à lire, à tester et à étendre.

L'idée a donc été de garder une carte simple, qui affiche uniquement les données essentielles, puis de laisser chaque décorateur ajouter sa propre couche d'affichage lorsque l'utilisateur active l'option correspondante. La vue crée la carte, regarde quelles cases sont cochées, et applique les décorateurs un par un. L'utilisation de Supplier<CardDecorator> a rendu ce mécanisme encore plus propre : on ne crée un décorateur que si l'utilisateur l'a demandé.

Chaque option repose sur une logique simple dans le modèle, mais reste indépendante du reste de l'affichage.
La coloration utilise l'expérience totale du candidat, les détails d'expérience exploitent les descriptions déjà fournies par le modèle, les compétences rares sont calculées dans *ApplicantList* à partir de la fréquence des compétences dans l'ensemble des profils, et les red flags proviennent des indications retournées par la méthode getRedFlags() dans *Applicant*.
La vue se contente d'afficher ce qu'on lui donne, sans connaître la mécanique interne qui permet d'obtenir ces informations.

Cette organisation offre deux avantages concrets. D'abord, ajouter une nouvelle option visuelle devient trivial : il suffit d'écrire un décorateur et de l'enregistrer dans la vue, sans toucher à *ApplicantCard* ni au reste de l'interface. Ensuite, les options peuvent être combinées librement : un candidat peut être coloré, afficher ses red flags, montrer ses compétences rares, tout en révélant ses expériences détaillées, le tout sans aucune dépendance entre les décorateurs.

Au début, on doutait un peu : *est-ce qu'un décorateur a vraiment sa place dans la vue ?* On l'associe plutôt au modèle. Mais en pratique, ça s'est révélé parfait : la *ApplicantCard* reste propre et minimale, et chaque option d'affichage vient simplement se greffer par-dessus, sans rien modifier au code existant. On enrichit l'affichage quand on en a besoin (couleur, détails, red flags…), et on ne crée aucun décorateur tant que l'utilisateur ne l'active pas. Au final, on profite des avantages du pattern sans jamais toucher au modèle, et l'ajout d'une nouvelle option visuelle devient extrêmement simple.



*package view: La vue reçoit uniquement des données prêtes à afficher (**ApplicantViewData**), construit une carte simple (**ApplicantCard**), puis applique les décorateurs choisis pour enrichir visuellement le résultat.*

ApplicantViewData comme DTO de la vue

Pour compléter ce découplage entre la vue et le modèle, on a introduit une petite brique vraiment pratique : ***ApplicantViewData***. Ce n'est pas un candidat, ce n'est pas non plus un élément du modèle ; c'est juste un "paquet" préparé par le contrôleur, qui rassemble uniquement ce que la vue doit afficher. Grâce à ça, la vue n'a plus besoin d'aller fouiller dans les objets du modèle ni de comprendre comment une expérience est stockée ou comment une moyenne est calculée. Elle reçoit des données déjà prêtes, propres et cohérentes, et elle se contente de les afficher. Ça simplifie énormément l'interface et ça évite que des détails internes du modèle ne se propagent partout dans l'application.

Un petit mot sur le Builder

L'ApplicantBuilder n'est pas un pattern qu'on a réellement réalisé dans le projet, mais on l'a fait évoluer pour répondre à nos besoins réels. Au départ, il ne récupérait que les compétences ; on l'a ensuite modifié pour reconstruire aussi toutes les expériences du candidat, ainsi que leurs mots-clés (ce que le candidat a fait pendant son expérience).

Concrètement, on s'appuie sur la structure du YAML où chaque expérience est un bloc nommé par l'entreprise, contenant start, end et une liste keywords. Le builder parcourt simplement ces entrées, crée une instance de **Experience** pour chacune, puis l'ajoute au candidat dans la liste de ses expériences.

Éthique

Neutralité, non-discrimination et choix de conception

Dès la conception, nous avons cherché à produire un outil aussi neutre que possible. Contrairement à un recruteur humain soumis à des biais inconscients, notre algorithme applique strictement les mêmes règles à tous les candidats.
Pour garantir cela, nous avons fait le choix de ne **traiter que des données techniques** et totalement objectives : compétences, expériences, mots-clés, durées. Les informations personnelles comme le nom ou le parcours scolaire ne jouent aucun rôle dans la sélection. L'algorithme ignore même l'ordre de chargement des CV : un candidat importé en dernier est évalué exactement comme les autres. Cela limite des biais classiques comme la "fatigue décisionnelle" observée dans les recrutements humains.

Cette neutralité ne signifie pas que l'algorithme est parfait, mais elle assure que chaque CV est traité avec les mêmes règles, quelles que soient les caractéristiques personnelles du candidat.

Notre algorithme ne peut pas donc être accusé de discrimination directe puisqu'il n'utilise aucune variable protégée par la loi, que des données objectives.

Comment éviter de rater des bons profils ?

Le cœur de l'outil repose sur des stratégies de sélection modulables. Le recruteur choisit les compétences qu'il recherche ; l'algorithme sélectionne les profils qui correspondent à ces critères et les classe ensuite selon la stratégie active. Par défaut, les candidats sont triés par la moyenne de leurs compétences sur les skills demandées (du meilleur au pire) , ce qui met en avant ceux qui maîtrisent le plus les technologies recherchées.

Mais nous avons aussi ajouté un tri alternatif basé sur les années d'expérience pour permettre au recruteur de mieux distinguer juniors, intermédiaires et seniors (vert –> seniors: plus de 10 ans d'éxperience; jaune –> intérmediaires:plus de 5 ans d'experience ; rouge →juniors: moins de 5 ans d'experience).

Ce changement de tri ne modifie pas la sélection, mais il change ce qui est mis en avant : un junior très compétent peut rester visible même si le tri favorise les profils expérimentés.

Nous avons également introduit la notion de compétences rares. L'idée est simple : certaines skills n'apparaissent que chez un ou deux candidats, et sans un mécanisme dédié, ces profils atypiques pourraient passer inaperçus. L'algorithme les met donc en valeur via une option d'affichage, afin d'éviter de "louper" des profils précieux mais minoritaires.

Ces choix vont tous dans la même direction : éviter qu'un modèle trop rigide ne filtre à tort des candidats prometteurs et talentueux.

Gestion des risques

Nous avons aussi intégré une détection automatique de signaux potentiellement sensibles, comme les expériences très courtes ou les trous dans le CV. Cette fonctionnalité a pour but d'aider un recruteur à repérer plus vite des éléments à surveiller, notamment dans un contexte où certaines entreprises reçoivent des dizaines de milliers de CV.

Mais ces indicateurs ont des limites évidentes :un algorithme peut interpréter un congé parental, une maladie, une immigration, une réorientation ou une année sabbatique comme un "trou" problématique, alors qu'un humain comprend le contexte. C'est précisément dans ces situations qu'un système automatisé peut se tromper.

Pour éviter que l'algorithme ne devienne trop rigide ou qu'il porte un jugement définitif sur un candidat, nous avons donc fait un choix clair : toutes les analyses avancées (mise en évidence des red flags, coloration selon l'expérience, affichage des compétences rares) restent entièrement optionnelles. Le recruteur peut activer ou désactiver ces éléments à tout moment et revenir, s'il le souhaite, à un affichage totalement neutre, limité aux scores et aux années d'expérience.

L'ajout de l'affichage détaillé des expériences va dans le même sens : même lorsqu'un red flag apparaît, le recruteur peut en comprendre le contexte en lisant précisément ce que le candidat a fait ou vécu. L'outil met simplement en lumière des signaux ; l'interprétation, elle, reste dans les mains de l'humain.

En d'autres termes, le logiciel accompagne la décision, **mais ne la remplace jamais**.

Un outil qui accompagne la décision, sans jamais la remplacer

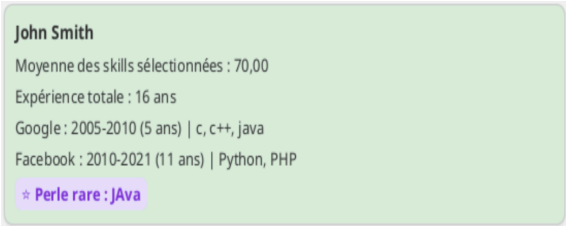
Au moment de la mise en service, nous avons constaté que le logiciel remplit exactement le rôle que nous voulions lui donner :
mettre en avant les profils cohérents avec les compétences recherchées, faire ressortir des profils atypiques via les compétences rares, et signaler les éventuelles anomalies de parcours via les red flags, tout en laissant systématiquement la décision finale au recruteur.

Nous sommes pleinement conscients que la complexité d'un parcours humain dépasse de très loin ce que peut analyser un programme. Notre outil n'est pas un filtre définitif : c'est un support de tri, une aide visuelle, un moyen de structurer rapidement un ensemble de CV. Il ne remplace ni le jugement humain, ni l'entretien, ni la compréhension du contexte personnel ou professionnel d'un candidat. On suit donc la même logique que les outils utilisés dans les grandes entreprises : l'algorithme aide à traiter un grand nombre de candidatures (donc faciliter des tâches pénibles), mais la décision reste toujours humaine.

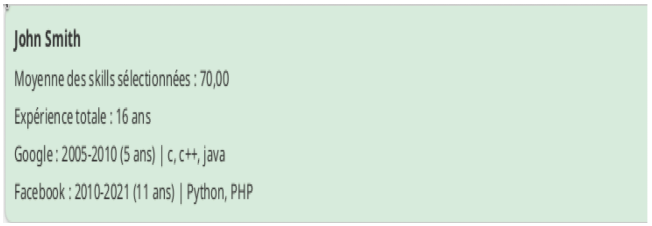
Tests

En parallèle des tests automatisés, qui nous ont assuré que le code fonctionnait correctement sur le plan technique, nous avons beaucoup misé sur des **tests manuels** pour vérifier que l'application réagissait réellement comme un recruteur l'attendrait. C'est souvent dans ces essais manuels que les vrais problèmes apparaissent, non pas des erreurs d'exécution, mais des comportements incohérents.

Un cas intéressant que nous avons trouvé concerne la détection des compétences rares. Pendant les tests, on s'est rendu compte qu'un même langage pouvait être traité comme plusieurs compétences différentes selon la façon dont il était écrit dans le CV ("JAVA", "java", "JaVa"…). L'algorithme se retrouvait alors à considérer ces variantes comme des compétences distinctes, ce qui faussait totalement le résultat (voir en bas). Cette anomalie, invisible dans les tests automatiques, nous a forcés à normaliser (donc insensibles à la casse) toutes les compétences avant de les analyser.



avant normalisation



après normalisation