Contreras Baptiste **p1809436**
Dumont Mathéo **p1602557**

Rapport de conception

Introduction

Ce projet permet de rechercher et de classer des candidats en fonctions de critères saisis par l'utilisateur et en appliquant des stratégies de trie prédéfinies. Nous avons implémenté une interface basique, plusieurs stratégies de base ainsi qu'un historique des recherches effectuées, qui peut être sauvegardées.

Design pattern

GRASP

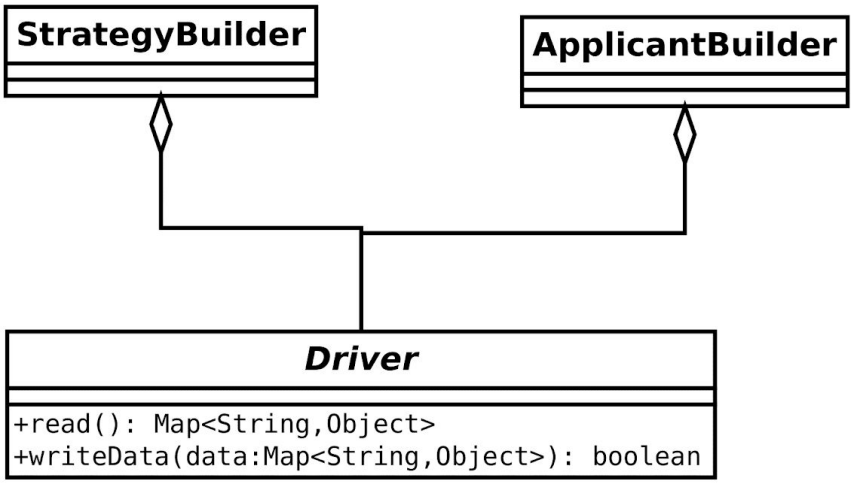
Un exemple de classe expert dans notre projet est la classe Applicant. Lorsqu'on veut savoir si un Applicant à de l'expérience professionnelle dans un domaine, on s'adresse directement à lui et non à *l'ApplicantList*. (pour faire un parallèle avec l'exemple du cours, l'Applicant serait un livre et *l'ApplicantList* la bibliothèque).

Un exemple de Forte Cohésion est notre package *SearchRequest.* Au lieu d'appliquer les stratégies sur une liste d'*Applicants* avec toutes les informations impliquées dans le *Controller*, ce travail est relégué à la classe SearchRequest qui non seulement centralise l'application des stratégies, mais permet aussi par son existence l'emploi de plusieurs stratégies sur un jeu d'*Applicant* sans impacter le *Controller*.

Ainsi toutes les opérations en liens avec l'application de stratégies sont centralisées en un seul lieu, comme l'implémentation d'un historique de stratégies appliquées.

ApplicantBuilder regroupe *ApplicantBuilder* et *ApplicantListBuilder* découplant ainsi la création d'*Applicant* se fait par cette classe et représente le pattern Créateur, favorise un couplage faible.

Indirection



Nous utilisons de nombreux Builder dans notre application, que ce soit pour charger les Applicants, les Strategy ou d'autres objets. Le point commun de ces Builders est qu'ils ont besoin, à un moment, de faire un appel à un "élément" extérieur au programme. Cet élément peut être un fichier yaml, une base de donnée sql, une api web etc...
Pour garantir un faible couplage et faciliter l'ajout d'accès aux données, nous avons mis en place des Indirections.

Elles sont présentent sous forme de Driver. Chaque Builder utilise un driver, par exemple un *YamlDriver* pour interagir avec des fichiers Yaml ou encore un SqlDriver, pour accéder à une base de données. Tous ces Driver permettent non seulement d'être réutilisés entre les Builder et les autres objets qui ont besoin d'accéder à du stockage (cf *Momento*) mais aussi d'ajouter très simplement un nouveau système de stockage.
Les Builder peuvent utiliser un Driver pour la lecture et un autre différent pour l'écriture.

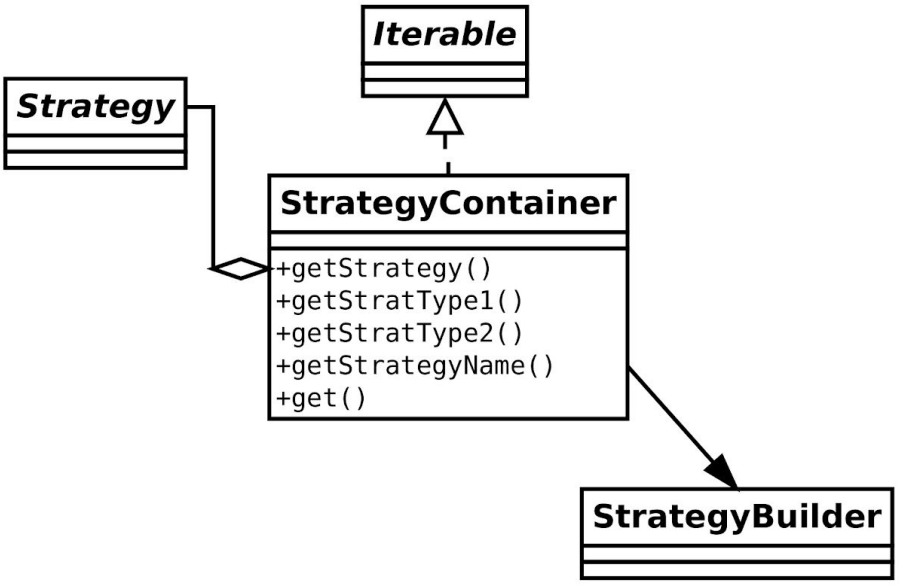
Builder

Notre application possède de nombreuses classes servant à la création d'objet plus ou moins complexe. Parmis ces "Builder", on retrouve L'*ApplicantBuilde*r chargé de créer les candidats, le *RequestBuilder* qui créer la requête de tri (cf *Chaîne de responsabilité*) et bien d'autre. Le point commun de ces Builder est l'utilisation d'indirection pour charger les données de façon transparente (cf I*ndirection*). L'avantage est que les objets de notre application n'ont pas besoin de se soucier de la création d'autre objet, qui est la tâche de nos Builder. On diminue ainsi le couplage et on décharge certaine classe.

Observer

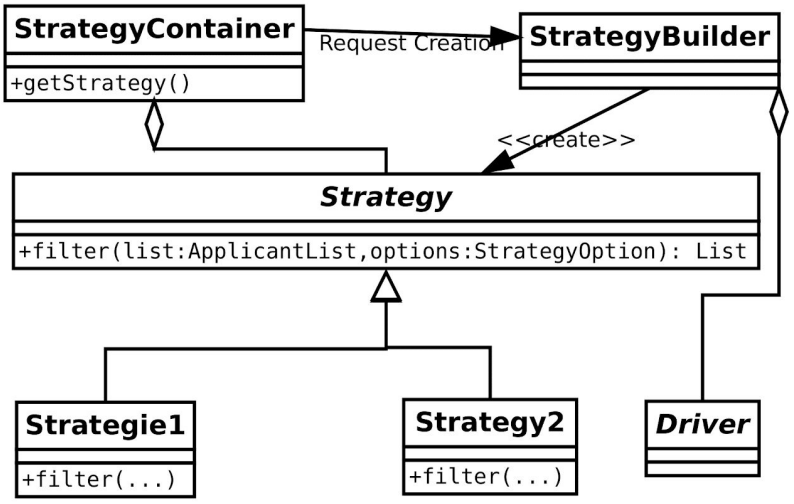
Le pattern *Observer* est utilisé pour notifier la vue de la fin d'un calcul et que les données résultantes sont disponibles pour l'affichage.
Le *Model* indique à la vue quelle modification il vient de faire (énumération d'action)
L'implémentation du pattern *Observer* facilite l'implémentation de plusieurs vues qui auraient accès au même *Model* car on ne passe pas par le *Controller* pour récupérer des informations. Plusieurs vues ayant le même *Model* pourraient être mises à jour simultanément.

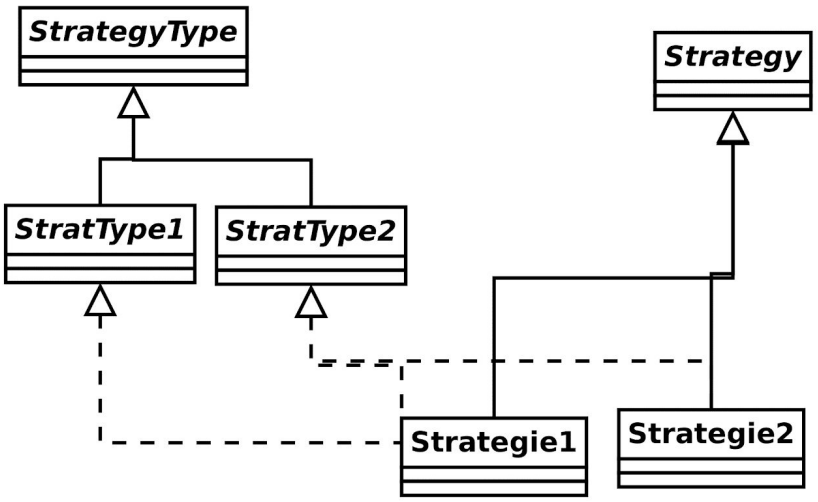
Wrapper



Notre projet contient plusieurs classes qui servent de wrapper de haut niveau. Par exemple *l'ApplicantList*, *SkillList* ou le *StrategyContainer* en font partie. Ces classes proposent de servir de Liste pour un type d'objet précis, mais en proposant des fonctionnalités plus spécifiques. Par exemple le *StrategyContainer* à des méthodes pour retourner des stratégies selon un critère donné. Ces Wrapper possèdent aussi un Builder permettant de construire leurs éléments. Grâce à ces Wrapper, on peut profiter de liste ayant des fonctionnalités propres à leur type d'élément.

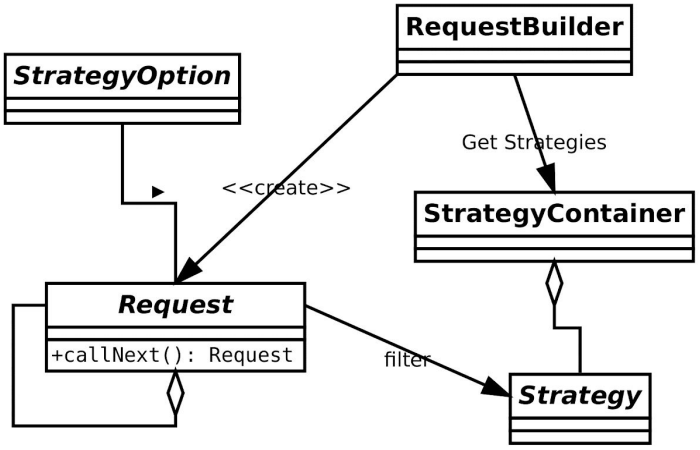
Strategy





Un des principaux design pattern utilisé pour notre projet est le design pattern Strategy. D'après le schéma UML ci dessus, on peut voir que chaque stratégie de trie dérive la classe "Strategy" qui requiert l'implémentation d'une méthode "filter". Cette méthode filter a pour but de prendre une liste *d'Applicant* et de la trier selon l'algorithme choisi. Ce design pattern est pertinent dans notre projet vu qu'il permet d'appliquer la bonne méthode filter sans avoir à se soucier du type de stratégie instancier. Cela permet de profiter du polymorphisme et simplifie l'ajout de future stratégie car il suffit d'implémenter la bonne interface et la méthode qui va avec. Une deuxième implémentation du design pattern strategy est présente dans notre projet. Les Stratégies peuvent implémenter une interface qui dérive StrategyType. Chaque StrategyType apparaîtra dans une combobox différente de la vue. Cela permet de définir dans quelle combobox iras chaque strategy et ensuite la Vue et le Model les utilisent pour l'initialisation. Bien évidemment, une Stratégie peut implémenter plusieurs types de StrategyType pour être dans plusieurs combobox.

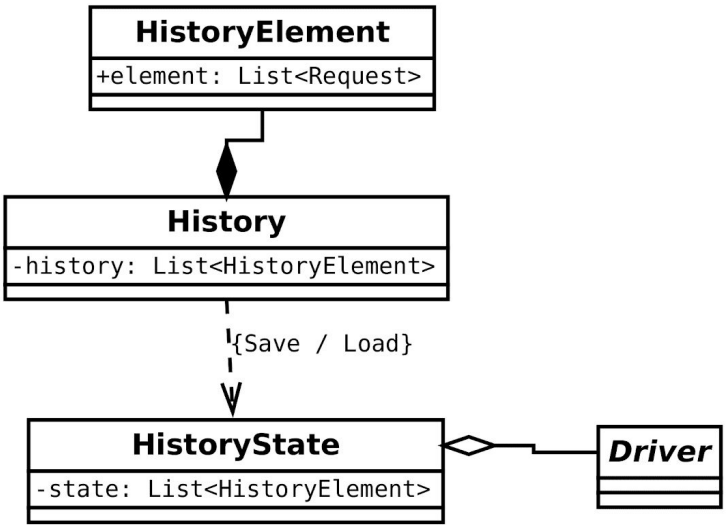
Chaîne de responsabilité:



Pendant la conception, nous avons cherché à maximiser la simplicité d'ajout de fonctionnalité. Dans cette optique nous avons mis en place le design pattern Chaîne de responsabilité pour les recherches de candidat. Nous avons représenté l'application de filtre(s) sur une liste d'*Applicant* par l'objet SearchRequest. Cet objet contient la liste de départ, la stratégie à appliquer sur cette liste avec des options, et le résultat. La requête (cf *SearchRequest*) connait aussi la prochaine requête à appliquer sur son résultat. De ce fait, l'ensemble des requêtes forment une chaîne qui appliquent leur stratégies au résultat de la requête précédente. Le résultat final se trouve dans la dernière requête. Ce système peut vite devenir compliqué à fabriquer si on à beaucoup de stratégie à appliquer en cascade pour une recherche, c'est pourquoi le *RequestBuilder* permet de construire facilement la chaîne de requête, en fonction de ce que l'utilisateur a choisis. Le builder retourne la chaîne de requête construite et prête à être insérée dans l'historique des recherches.
Une des raisons qui nous a fait choisir ce design pattern est la possibilité de mettre en place un historique de recherche (cf *Momento*) comme mentionné plus haut. Un élément de l'historique correspond à la chaîne des requêtes qui a été appliqué lors d'une recherche précédente. Cela nous permet de l'afficher sur l'interface et de rejouer la recherche sans recalculer le résultat.

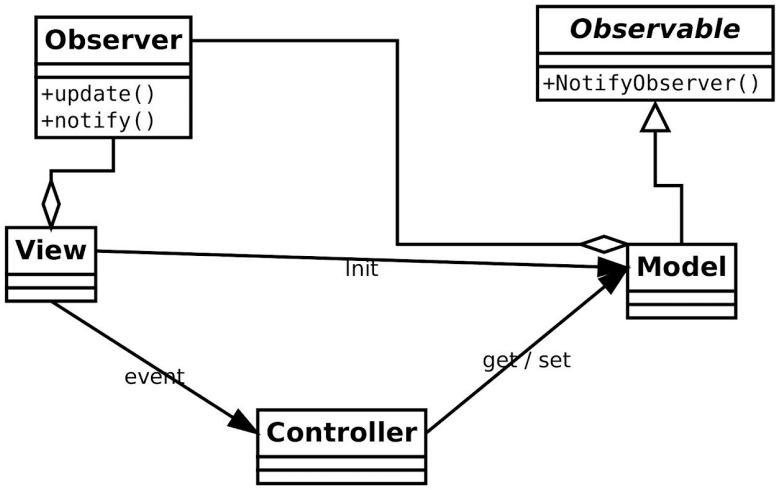
D'autres pattern peuvent être appliqués à la place comme Interpréteur mais nous ne l'avons pas fait car nous pensons que ce design pattern serait plus adapté dans le cas ou on aurait des recherches plus paramétrables (ex : l'utilisateur peut modifier beaucoup de valeurs pouvant impacter les filtres voir ajouter un peu de logique sur comment les appliquer, comme dans le cas suivant, Filtre1 ET Filtre2 OU FILTRE3 SI valeur1 > 5) . Ayant choisis de faire un système de recherche un peu plus simple, ce design pattern aurait été un peu compliqué par rapport à nos besoins et moins facile à faire évoluer.

Momento



Pour la sauvegarde et la restitution de l'historique, nous avons mis en place le design pattern Momento. l'historique contient des éléments qui eux même contiennent une liste de requête précédemment utilisée. Lorsqu'on veut le sauvegarder, l'historique créer un "momento" contenant la liste de ces éléments. Le "momento" va mettre en forme les éléments pour les envoyer à un Driver (qui représente dans ce cas le Caretaker), et qui va par la suite, sauvegarder les données. Le driver peut être de n'importe quel type, c'est une classe qui sert à lire ou à écrire en dehors de l'application, peut importe ce qu'on lui demande, et pour lui les données sont transparentes. Pour restituer l'historique, le momento utilise le driver pour lire les données brut, les mets en forme et redonne la liste d'élément à l'historique. Cela permet de sauvegarder et de charger l'historique en empêchant sa modification.

MVC



Comme imposé par le sujet, notre projet respecte entièrement MVC.

La vue accède au *Model* pour l'initialisation et pour récupérer des informations grâce à des getter lorsqu'elle est notifié d'un changement (cf *Observer*). Les événements appellent des méthodes du controller.

Le *Controller* fait le lien entre la vue et le model. Chacune de ces méthodes, une fois appelée par un événement de la vue, appellent une ou plusieurs méthodes du *Model* pour appliquer des changements.

Le *Model* s'occupe de la partie métier, une fois une modification appliquée, il notifie la vue grâce à ces *Observer*. Il est aussi chargé de stocker des Objets, qui peuvent être utilisés par le *Controlleur*, la vue (pour se mettre à jour) ou par d'autres éléments du *Model* lors de processus métier.

Éthique de notre stratégie optimisé

Le principal choix éthique à faire pour notre algorithme est de savoir si on privilégie les candidats avec de l'expérience professionnelle ou non. En effet, notre algorithme ne connaît ni l'origine du candidat, ni son sexe et encore moins sa religion, il sera donc neutre sur ces points sans que nous ayons besoin de faire quelques choses. Du coup, ayant éliminé ces critères "critique" (un algorithme discriminant sur un de ces points n'est pas acceptable, éthiquement parlant ou non), il nous reste la responsabilité de choisir entre l'expérience professionnelle ou personnelle. Comment peut-on être sûr qu'une personne ayant 10 ans de java en entreprise est meilleur qu'un candidat ayant fait plusieurs projets personnel conséquent ?

Nous avons fait le choix de privilégier l'expérience professionnelle et cela rend notre algorithme imparfait.(Notre algorithme applique un coefficient bonus pour les skills utilisés en entreprise). Pour réduire cet effet "injuste", il faudrait qu'on prenne en compte plus d'informations sur les candidats comme leur projet personnel, s'ils ont travaillé sur un ou plusieurs projet dans leur vie etc. Un autre champs qui pourrait rentrer en compte dans l'équation est l'âge mais la aussi on aurait un problème d'éthique, qui favoriser en cas de candidature plus ou moins identique ? et comment être sûr qu'on ne fait pas le mauvais choix ? Pour conclure sur la partie éthique, le moyen de réduire les "inégalités" est de rendre l'algorithme le plus juste possible en lui fournissant le plus d'information "neutre" (des informations plus techniques que personnel) . Il y a plus de détails sur l'algorithme dans le fichier de test "StrategyTest.java".

Tests

Nous avons réalisés des tests principalement sur nos *Strategies* et la chaîne de responsabilité que représente *SearchRequest* et *SearchRequestBuilder* étant donné que cela est le coeur de notre application.
Nous avons aussi testé certaines fonctions du *Controller*.

Les tests des Stratégies vérifient en première partie que les comportements optimaux sont fonctionnels; donc que telle stratégie avec une *ApplicantList* et des *Options* contenant nos expériences et entreprises retournent bien le résultat attendu. Ensuite, nous testons les cas où certains champs dans Options sont vides, comme *Skill List* ou *CompanieList.*

Pour *SearchRequest* et *SearchRequestBuild*, c'est la même chose, d'abord nous vérifions que les *SearchRequest* sont conformes aux attentes en fonction des stratégies et informations données en entrée.
Pour ensuite tester la chaîne de responsabilité proprement dite dans différentes situations, donc comme pour le cas des Stratégies, avec différents états des champs dans *Options*, ainsi qu'un nombre variant de stratégie à appliquer sur *l'ApplicantList* en départ.

Pour faciliter l'implémentation des tests, nous employons une classe Data qui construit pour nous les *DataSet* nécessaire tel qu'un jeu *d'Applicant*.
Pour tester le Controller, il est nécessaire de pouvoir effectuer des changement dans le même environnement que celui de fonctionnement. Un Mock est donc tout indiqué. C'est là qu'intervient la classe Mockingbird; elle instancie une classe abstraite view vide : l'affichage n'étant pas nécessaire pour tester le *Controller* ou le *Model* (le test de l'affichage se retrouve plus dans des tests systèmes ou d'intégration).

Avec ce Mock, nous pouvons observer facilement la répercussion d'appels sur *Controller* et sur le *Model*. Et c'est ce qui est fait dans *ControllerTest*.