Université de Nice Sophia Antipolis
Faculté des Sciences
Maîtrise d'informatique 2002 - 2003

Gabriel ZERBIB
Lucas CHARBIT
Jérôme GAHIDE
Xavier GALBOIS

JavInspector

Travail d'étude et de recherche

Encadrants : Pierre CRESCENZO
Michel GAUTERO
Philippe LAHIRE

SOMMAIRE

INTRODUCTION

Lorsque nous avons du choisir le sujet de notre Travail d'Etude et de Recherche, nous avons longuement hésité. Nous étions tous d'accord pour privilégier les sujets de développement. Nous trouvions intéressant d'avoir à réaliser une application dans son ensemble.

Nous avons donc choisi le sujet : « JavInspector un inspecteur d'applications Java ». En voici une description.

JavInspector devra être une application permettant l'inspection non intrusive de classifieurs (classes et interfaces) Java ainsi qu'un logiciel fini mais extensible - entièrement écrit en Java - et fonctionnant aussi bien sous Windows que sous Linux. JavInspector devra de plus être bien documenté (Javadoc, entièrement écrite en anglais); et le code de l'application devra être de qualité et lisible. L'interface graphique devra être ergonomique, intuitive et simple d'utilisation pour permettre à un utilisateur de se servir de JavInspector de façon pratique.

Si ces conditions sont remplies JavInspector pourra être diffusé sous Licence GNU LGPL http://www.fsf.org/licenses/lgpl.html .

JavInspector doit intégrer au terme de son développement final, deux fonctionnalités principales : l'analyse de fichiers .class et l'analyse de fichiers .java. Voici une description de ces deux fonctionnalités.

Premièrement, si l'on fournit à l'application un fichier .class celle-ci devra permettre de charger le classifieur en mémoire (si cela n'est pas déjà fait) puis d'afficher graphiquement diverses informations le concernant : nom, méthodes, constructeurs …

Deuxièmement, si l'on fournit à l'application un fichier .java celle-ci devra procéder à l'analyse syntaxique du fichier et afficher des informations similaires à celles fournies par la première fonctionnalité

Nous devions implémenter la première de ces deux fonctionnalités, la seconde étant considérée comme un bonus si elle était réalisée. Malheureusement, par manque de temps, nous n'avons pas entamé la réalisation de la deuxième fonctionnalité.

Par la suite nous allons décrire de manière détaillée chaque partie de l'architecture du logiciel. Nous allons également expliquer, sous forme de tutoriaux fournis en annexe comment étendre ou modifier certaines parties de l'application.

Nous expliquerons aussi quels ont été les problèmes de conception rencontrés et les solutions apportées pour les résoudre.

Aparté

Dans la suite du rapport nous allons fournir des diagrammes de classes pour expliquer les différents modules de l'application.

Voici les différentes notations utilisées.

A  B : désigne l'héritage entre classes (A hérite de B)

A  B : désigne la composition multiple entre classes (A possède une ou plusieurs instance de B et B possède une ou plusieurs instance A)

A  B : désigne la composition simple A possède une ou plusieurs instance de B.

I Architecture de l'API de l'analyse de classifieurs

I.1 Présentation

L'API servant à l'analyse de fichier est le cœur de l'application, c'est elle qui fait office de modèle dans le «design pattern » MVC (expliqué plus loin).

C'est donc grâce à elle que sont récupérées toutes les informations nécessaires aux différentes vues de notre application.

I.2 Diagramme de classes

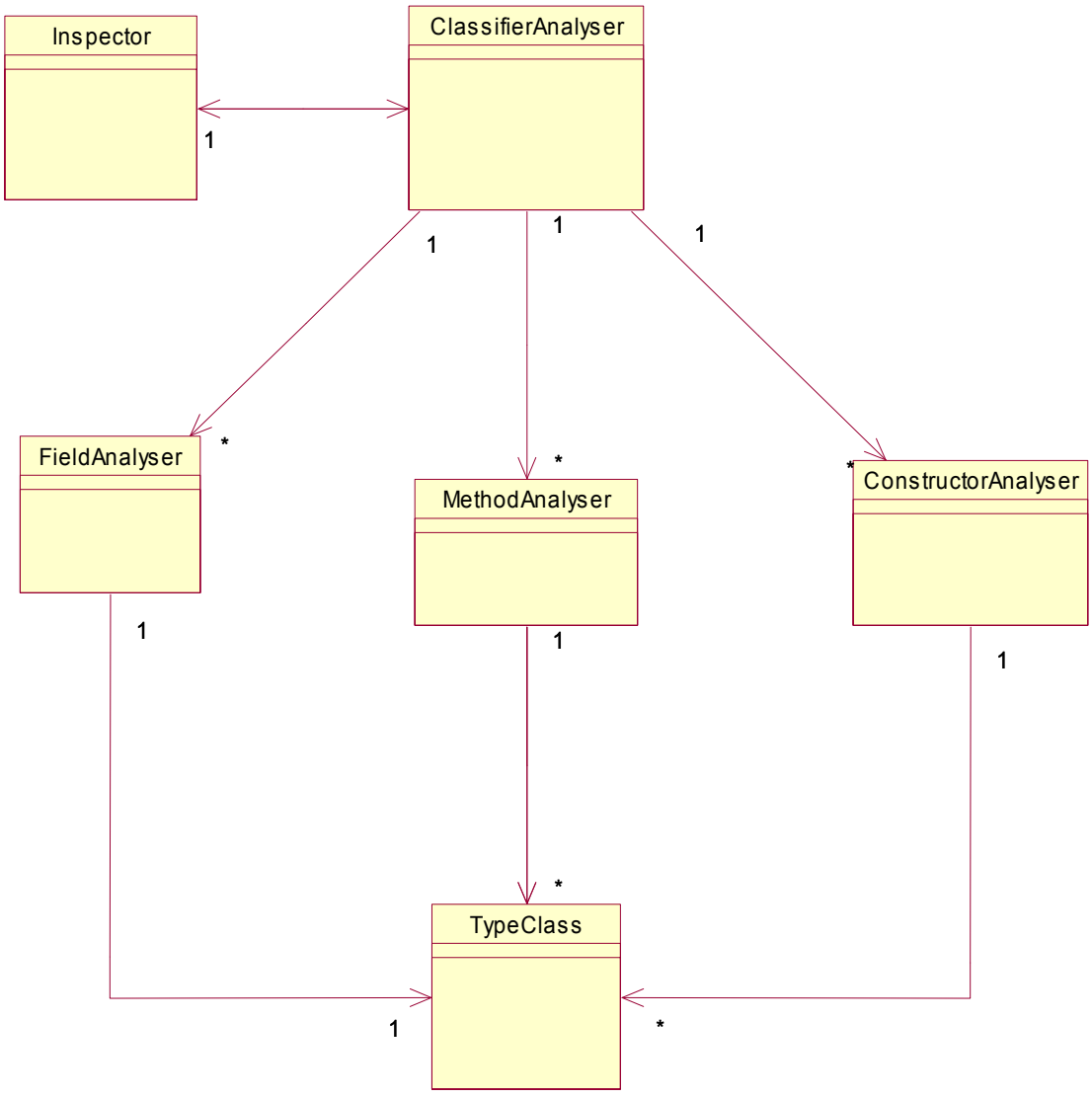


Figure I.2—1 : Diagramme de classe de l'API

Comme on peut le voir sur le diagramme de classe ci-dessus l'API servant à l'analyse de classifieurs se compose des classes suivantes :

`Inspector` : cette classe contient tous les `ClassifierAnalyser`.
`ClassifierAnalyser` : cette classe contient toutes les méthodes utiles pour l'inspection de classifieurs Java.
`MethodAnalyser` : cette classe contient toutes les méthodes utiles pour l'inspection de méthodes Java.
`ConstructorAnalyser` : cette classe contient toutes les méthodes utiles pour l'inspection de constructeurs Java.
`FieldAnalyser` : cette classe contient toutes les méthodes utiles pour l'inspection de champs Java.
`TypeClass` : classe intermédiaire qui sert à représenter une classe avec son nom complet (dans le cas ou celle-ci est un tableau) et sa classe. Exemple : si notre classe est `String[][]` le nom `TypeClass.name` vaudra `String[][]` et `TypeClass.cl` vaudra `String`.

I.3 Choix d'implémentation

Nous avons décidé de bien séparer toutes les données de l'interface graphique, c'est pourquoi notre API est un module à part, qui pourra être réutilisé indépendamment de l'interface graphique, par tout développeur désireux de s'en servir.

Nous avons séparé l'analyse des méthodes (methods), des champs (fields) et des constructeurs (constructors). Cela nous a permis d'être plus précis dans l'analyse de ces données et d'avoir plus d'informations sur celles-ci.

En effet nous avions décidé au départ de récupérer toutes les informations utiles à un classifieur dans la classe `ClassifierAnalyser` mais il est apparu que celle-ci contenait beaucoup trop d'informations redondantes et difficiles d'accès.

Toutes les informations sur un classifieur sont récupérées à l'aide du package `java.lang.reflect` de la JDK 1.4.

Il aurait été possible de trouver plus d'informations sur les classes (comme par exemple les initialiseurs) mais il aurait fallu pour cela « parser » le byte code Java à l'aide d'un outil comme Kopi.

I.4 Fonctionnalités

L'API permet de :

Récupérer toutes les classes héritées et toutes les interfaces implémentées par un classifieur si celui-ci est une classe.
Récupérer toutes les « super interfaces » d'un classifieur si celui-ci est une interface. Les « super interfaces » sont toutes les interfaces mères du classifieur ainsi que toutes les interfaces mères des interfaces mères du classifieur et ce récursivement jusqu'à ce que celles-ci n'aient plus d'interfaces mères.
Récupérer tous les champs (d'instances et statiques), toutes les méthodes (d'instances et statiques), et tous les constructeurs d'une classe.
Calculer les valeurs des champs statiques si ceux-ci sont des types primitifs ou des chaînes de caractère (String).
Récupérer toutes les classes et interfaces internes à un classifieur (ainsi que les classes anonymes qui auront comme nom « `nom_de_la_classe$*` », `*` étant un entier entre 1 et le nombre de classes anonymes).
Récupérer le nom complet d'une classe ainsi que son éventuel package d'appartenance.
Récupérer toutes les exceptions lancées par les constructeurs et les méthodes
Récupérer le nom complet des méthodes, des champs et des constructeurs
Récupérer tous les types utilisés dans les signatures des méthodes et des constructeurs ainsi que tous les types des champs.

II Module de chargement de classes

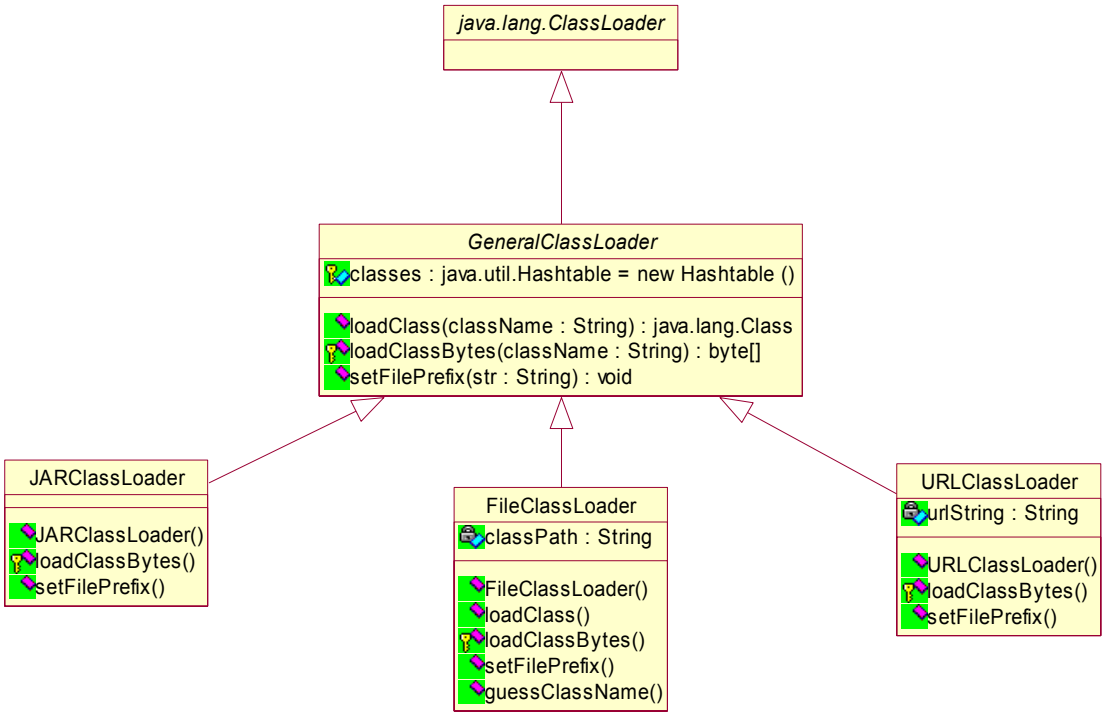
Pour que l'API d'analyse de classifieur puisse fournir des informations pertinentes il est nécessaire de charger le classifieur qui nous intéresse. La méthode utilisée est la réflexivité de Java à travers la classe `java.reflect.Class`. Ainsi pour un classifieur donné, il faut obtenir l'objet `Class` correspondant. Heureusement Java fourni les méthodes pour charger les fichier `'.class'` dans la Java Virtual Machine.

Ainsi lorsque le classifieur que l'on souhaite analyser est dans le `CLASSPATH` alors le chargeur de classe de Java suffit (`java.lang.ClassLoader`). Mais lorsque l'on souhaite analyser un classifieur qui n'est pas dans le `CLASSPATH`, le problème se complique car Java n'est pas capable de charger un tel classifieur

La solution est donc de créer notre propre chargeur de classifieur, afin de pouvoir paramétré le `CLASSPATH` pendant l'exécution du JavInspector.

Pour commencer nous détaillerons l'architecture choisie, puis nous expliquerons les choix d'implémentation et les problèmes que nous avons rencontrés. Enfin nous parlerons des résultats et des fonctionnalités obtenues ainsi que des extensions possibles

II.1 Diagramme de Classes



II.2 Choix d'implémentation

Le principe de l'architecture choisie est simple. Le cœur du chargeur de classe est la méthode `loadClass(String className)` du `GeneralClassLoader` ; son rôle est d'essayer de charger la classe avec tous les moyens qu'elle possède :

Elle regarde dans son cache si la classe a déjà été chargée.
Elle demande à `java.lang.ClassLoader` d'essayer de la charger.
Elle essaie de charger le classifieur grâce à la méthode `loadClassBytes(String className)`. Cette méthode est définie dans les trois classes filles de `GeneralClassLoader`. Ainsi selon l'instanciation de l'objet (dépendant du type de l'emplacement du fichier) un traitement différent est effectué.

Lors du chargement par `loadClassBytes(String className)` le chargeur de classifieur se contente de récupérer le tableau d'octet de 'byte code' et effectue un appel à la méthode `defineClass(byte[])` de `java.lang.ClassLoader`.

L'avantage de cette implémentation est la flexibilité qui nous a permis de programmer facilement le chargement de classes à partir de différentes sources : JAR ou URL.

II.2.1 Implémentation du chargeur de classes à partir d'un fichier local

Lors de la conception de l'application nous avons décidé qu'un fichier pourrait être chargé à partir de n'importe quel répertoire, tout en évitant d'avoir à spécifier le répertoire de base des packages. Le contraire obligerait l'utilisateur à saisir le chemin de base des packages et ensuite la classe qu'il veut charger dans cette arborescence.

Le problème qui est survenu lors de notre implémentation est la résolution des dépendances « inter package ». En effet pour pouvoir charger une classe dépendante il faut connaître le répertoire de base des package.

II.2.1.1 Première solution :

La première solution qui nous est apparue était de charger la classe puis de récupérer son nom complet (avec le nom du package) et ensuite d'en déduire le répertoire de base des packages. Cependant cette solution est devenue vite inadéquate car elle fonctionnait qu'en cas de dépendances liées à la composition. Dans le cas de dépendances d'héritage, le `ClassLoader` de Java doit résoudre les dépendances avant de charger la classe et comme il ne peut pas la charger on ne connaît pas son package et il est donc impossible de la charger.

II.2.1.2 Deuxième solution

Les dépendances liées à l'héritage nous ont contraint à adopter une autre solution. La première qui nous est venue à l'esprit est d'analyser le byte code est d'en extraire le nom de la classe ainsi que son package. Cependant cette solution, apparemment simple dans l'idée, s'est révélée trop coûteuse en temps à cause de la structure des fichiers `'.class'`.

II.2.1.3 Troisième Solution

La dernière solution que nous avons trouvée est moins efficace mais réalisable. Elle est moins efficace car elle essaie de deviner le nom de la classe. Elle consiste en une simple boucle qui réessaie de charger la classe avec, à chaque itération, un répertoire de base des packages un cran plus bas, jusqu'à arriver à la racine de l'arborescence.

II.2.2 Implémentation du chargeur du classifieur à partir d'une archive JAR

L'implémentation de cette fonction a été réalisée grâce au package `java.util.zip`. Cette fonction nécessite de spécifier l'archive JAR et de connaître le nom exact de la classe que l'on veut charger.

II.2.3 Implémentation du chargeur de classe à partir d'une URL

Le chargeur de classe permet de charger une classe à partir du web. Cependant cette fonction ne gère pas complètement les dépendances. En effet sa fonction principale est de charger une applet.

II.3 Résultats et fonctionnalités

Le chargeur de classifieurs à partir de fichiers fonctionne très bien pourvu que l'arborescence des packages soit correcte. Cependant il fonctionne aussi, sur des classifieurs ayant des dépendances vers un même package, quand ce package a été déplacé.

Pour l'archive JAR, les dépendances sont cherchées dans le JAR et dans le `CLASSPATH` mais pas dans un autre JAR qui n'est pas dans le `CLASSPATH`.

Pour le chargement à partir d'une URL, les dépendances sont cherchées dans le même répertoire que celui du classifieur chargé, mais aussi dans le `CLASSPATH`. Cette contrainte oblige les dépendances à se trouver dans le même package.

II.4 Possibilités d'extensions

Comme décrit précédemment, c'est la gestion des dépendances qui pose le plus de problèmes. Ceci est du à l'architecture de package de Java. Pour rendre notre chargeur de classe plus flexible, il est nécessaire d'élaborer une meilleure résolution des dépendances. Pour ce faire, il faudrait pouvoir ajouter plusieurs chemins de CLASSPATH, au cas où les dépendances ne se situeraient pas dans la même arborescence que le classifieur chargé. Par extension, il faudrait pouvoir chercher les dépendances dans d'autres archives JAR. Quant au chargement à partir d'une URL, il faudrait pouvoir rechercher les dépendances dans l'arborescence du classifieur analysé.

III L'interface graphique du JavInspector

La réalisation de l'interface graphique de JavInspector a posé plusieurs problèmes. Tout d'abord, l'encapsulation des données. Le but du JavInspector étant d'inspecter un classifieur, toutes les fonctions liées à l'analyse devaient être bien séparées du code de l'interface graphique. Deuxièmement, le choix des données, quelles données afficher, où les afficher ? Comment les faire interagir ? Enfin, et probablement le plus important, comment rendre l'application extensible ? Nous allons donc dans un premier temps voir les choix d'implémentation que nous avons faits, puis nous verrons comment nous avons essayé de répondre aux mieux à ces problèmes.

III.1 Choix d'implémentation

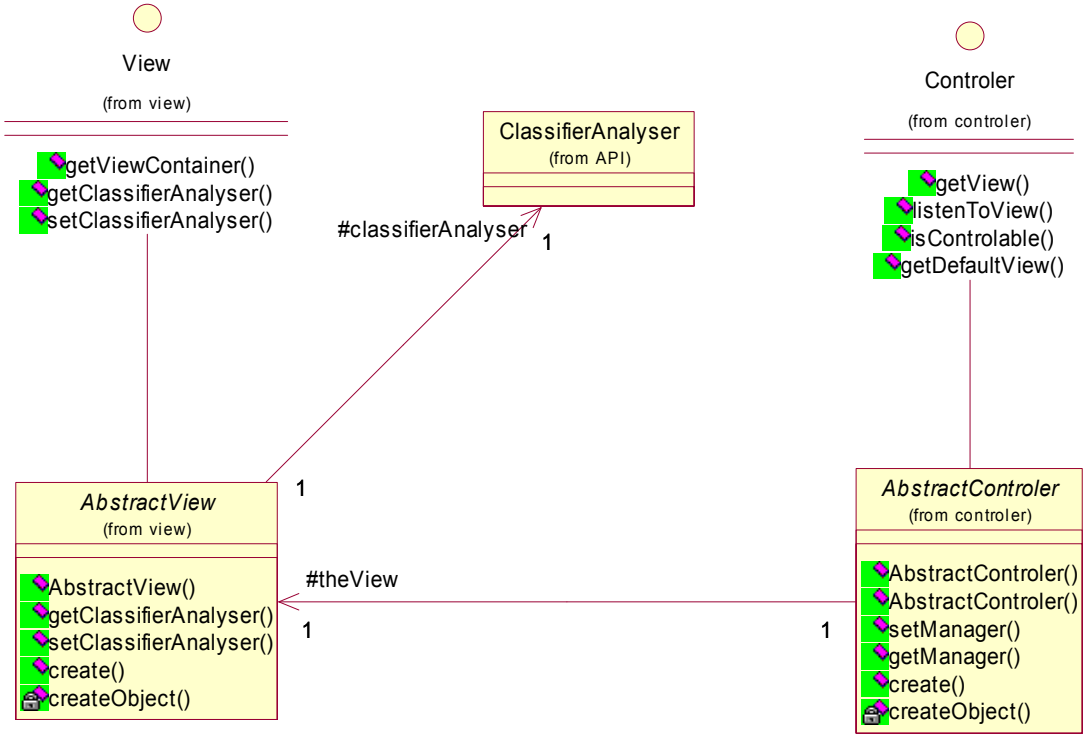
Pour choisir la façon de réaliser l'interface du JavInspector, nous avons tout d'abord cherché à bien identifier nos objets et les relations qui les lient. D'un côté, nous avions un ensemble de données sur un classifieur. Cet ensemble de données devait être bien séparé de l'interface graphique, pour être réutilisable. De l'autre côté, nous avions plusieurs objets graphiques, qui devaient récupérer leurs données dans notre premier ensemble, puis les afficher. De plus, ces objets graphiques devaient pouvoir communiquer entre eux et réagir à des événements déclenchés par les utilisateurs. Après discussion avec nos encadrants, il est apparu qu'il s'agissait d'un cas classique de programmation objets, pour lequel le modèle adéquate était le « design pattern » MVC (Model, View, Controler).

III.1.1 Design pattern MVC

Il s'agit d'un « design pattern » utilisé dans beaucoup d'applications graphiques et dont les origines remontent à Smalltak 80. Ce « design pattern » permet de préserver l'indépendance des composants, en séparant les données, leurs représentations et les événements. Un bon exemple d'implémentation de ce « design pattern » est le package Swing de Java.

Dans le cas du JavInspector, la situation est la suivante : notre ensemble de données qui est une instance de la classe `ClassifierAnalyser` est notre modèle (au sens MVC). Nos objets graphiques sont des vues, et à chaque vue est associée un contrôleur qui s'occupe de traiter tous les événements de cette vue.

Ainsi nous sommes arrivés au diagramme de classes suivant :



Cependant, notre implémentation présente plusieurs différences avec un MVC classique :

La première, est que dans notre cas, et contrairement au modèle MVC, notre modèle est purement statique. L'utilisation du « design pattern » Observer/Obervable entre une vue et son modèle (classique en MVC), nous a donc paru inutile.

De plus, nos vues n'ont pas de modèle propre, mais se partagent les données d'un même modèle (`ClassifierAnalyser`) que l'on pourrait considérer comme une grande base de données. Cela ne pose à priori pas de problèmes de conception car notre modèle n'est pas modifié par nos contrôleurs.

Le problème majeur que nous avons rencontré durant notre implémentation est le suivant : l'application crée un grand nombre de vues, et nous avions besoin de pouvoir les faire interagir entre elles (à travers les contrôleurs). Pour ce faire nous avons ajouté une nouvelle couche : les managers.

III.1.2 Les managers

III.1.2.1 Description

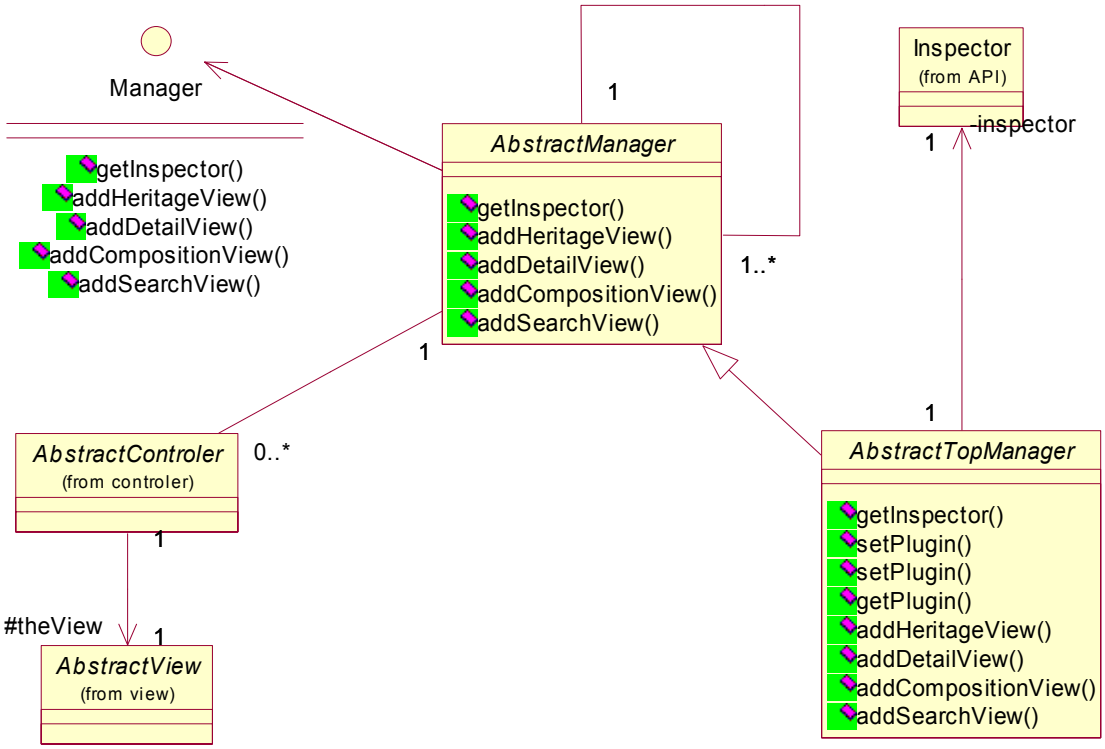
Un manager est un objet qui s'occupe de gérer un ensemble de contrôleurs et les vues qui leurs sont associées. Ce mécanisme se rapproche de celui mis en place par le « design pattern » Mediator. Plutôt que de permettre à tous les objets de communiquer directement entre eux, chaque objet a un manager auquel il demande de réaliser les actions nécessaires.

Le fonctionnement des managers est basé sur la délégation. Si un manager reçoit un événement de l'un de ses contrôleurs, il regarde si il est apte à le traiter. Si c'est le cas, alors il le fait, dans le cas contraire, l'événement se propage vers le manager supérieur.

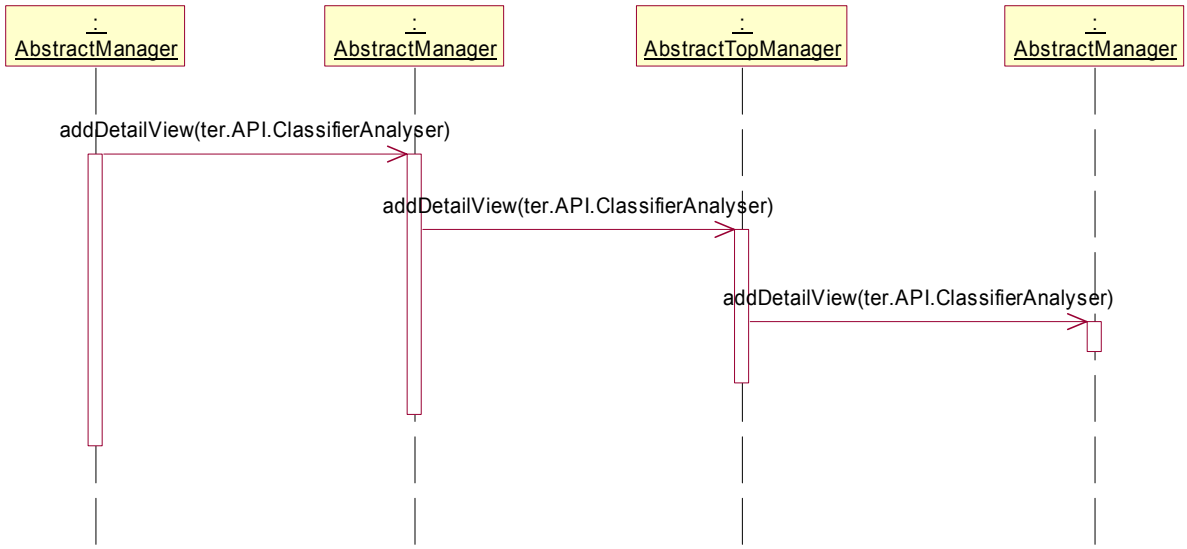
Il s'agit d'une architecture en couche, les contrôleurs sont dans des managers, qui peuvent eux-mêmes être dans des managers et ainsi de suite.

Il existe de plus un manager « central », qui est une instance d'`AbstractTopManager` qui est en fait la couche la plus haute de l'architecture.

Voici le diagramme qui montre notre architecture de managers :



Voici un exemple, le diagramme de séquence de la propagation d'un événement dans les différentes couches de l'architecture. L'événement part d'un manager, remonte jusqu'au `TopManager` qui le redirige vers le bon manager capable de donner suite a la requête.



Dans notre modèle, c'est l'instance d'`AbstractTopManager` qui a une instance d'`Inspector`. Chaque manager a donc accès à cette instance via une fonction de la classe abstraite. Grâce à la délégation, la demande se propagera jusqu'au `TopManager` et l'`Inspector` sera renvoyé.

Chaque manager a de plus un rôle graphique. Un manager est chargé d'organiser les contrôleurs et les vues qui leurs sont associées au sein de l'interface graphique. Il est de plus responsable de l'ajout et de la suppression de ses vues.

III.1.2.2 Fonctionnement (un exemple concret)

Nous allons maintenant regarder en détail comment nous avons implémenté notre architecture de manager dans JavInspector.

Dans notre cas, nous disposons de trois vues (expliquées en détail dans la prochaine section) sur un `ClassifierAnalyser`, une vue du graphe d'héritage, une vue du graphe de composition/utilisation et une vue détaillée. Nous avons choisi de regrouper la vue détaillée et la vue du graphe de composition/utilisation pour plus de clarté.

Notre configuration est donc la suivante :

Nous avons nos trois classes de vues :

La vue du graphe d'héritage : `TreeHeritageView`

La vue du graphe de composition/utilisation : `MyCompositionView`

Et la vue détaillée : `GraphDetailView`

Puis nos classes de contrôleurs qui écoutent ces vues :

`TreeHeritageControler`

`MyCompositionControler`

`GraphDetailControler`

Nous avons ensuite notre manager central, nous avons décidé de séparer nos deux groupes de vues avec un panneau coupé, ce qui explique son nom.

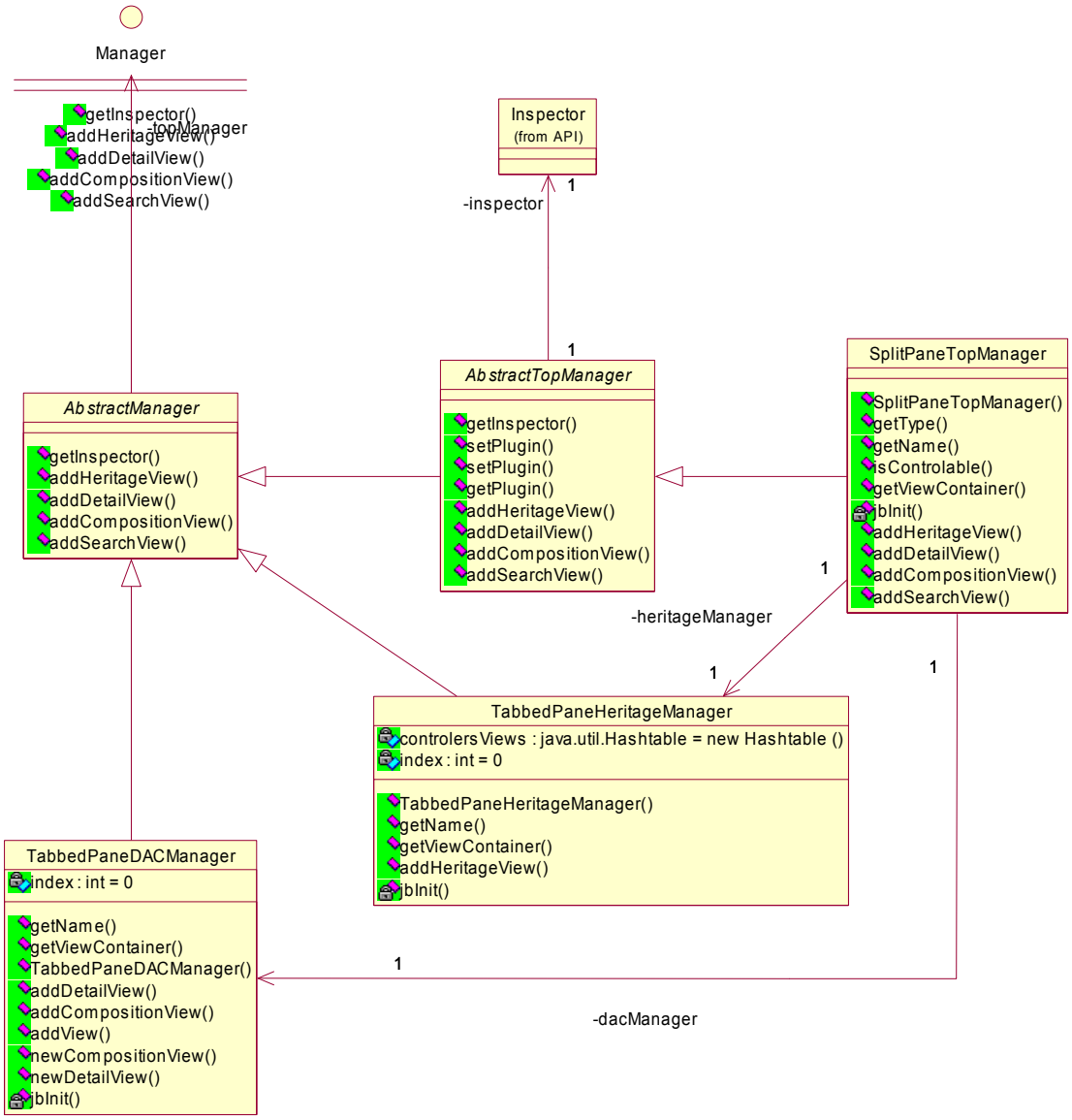
`SplitPaneTopManager`

Viennent enfin nos deux sous managers :

`TabbedPaneHeritageManager`, qui groupe les vues des graphes d'héritages dans des onglets

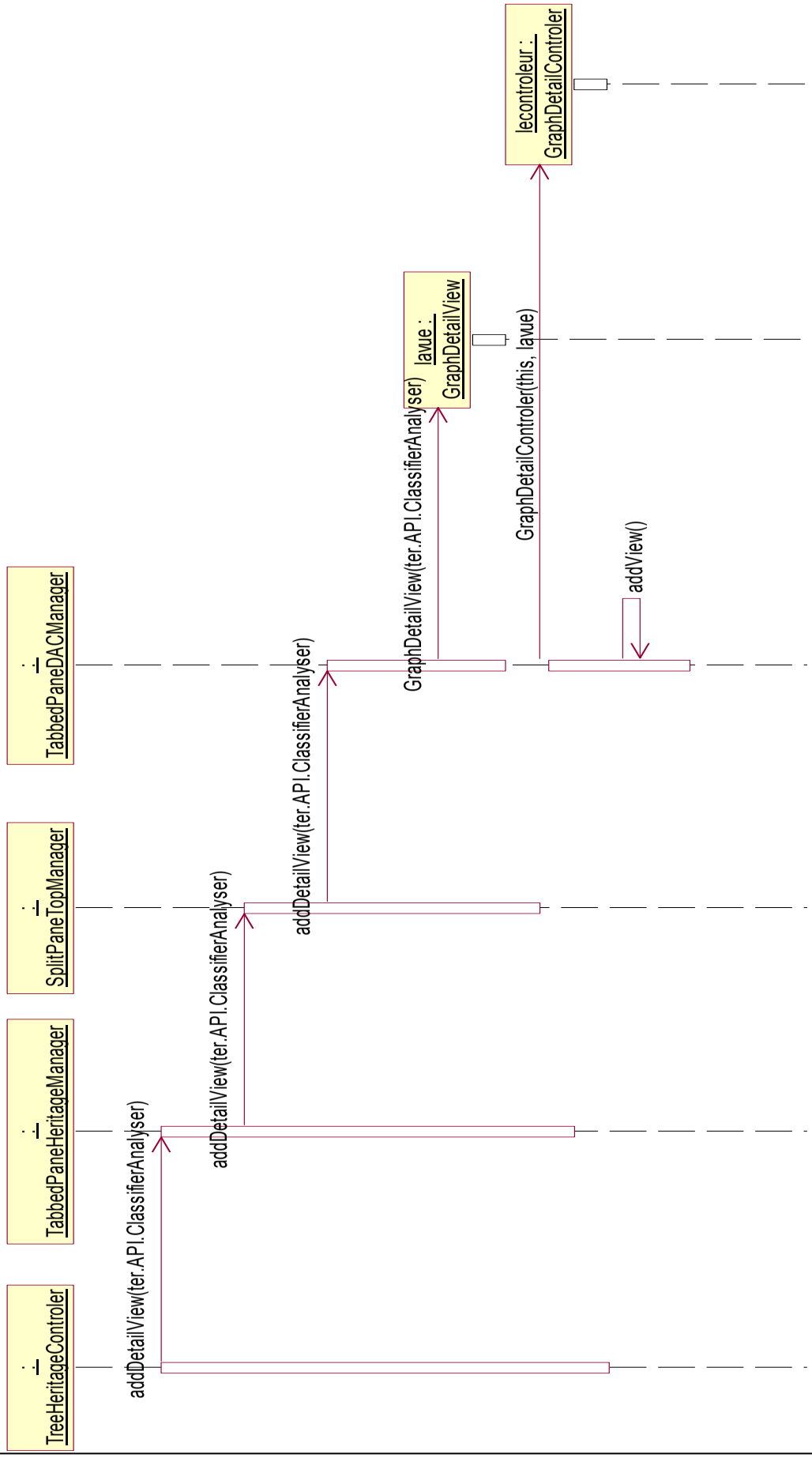
`TabbedPaneDACManager`, qui groupe dans un onglet la vue du graphe de composition/utilisation et la vue détaillée d'un `clasifierAnalyser`.

Pour résumer, voici le diagramme de classe de notre architecture :



Voici maintenant un exemple illustrant la délégation entre les managers. Il s'agit du diagramme de séquence de l'événement suivant :

L'utilisateur clique sur une classe du graphe d'héritage et demande l'affichage de son graphe de composition/utilisation et de sa vue détaillée.


TreeHeritageControler
TabbedPaneHeritageManager
SplitPaneTopManager
TabbedPaneDACManager
lecontroleur : GraphDetailControler
lavue : GraphDetailView
addDetailView(ter.API.ClassifierAnalyser)
addDetailView(ter.API.ClassifierAnalyser)
addDetailView(ter.API.ClassifierAnalyser)
GraphDetailView(ter.API.ClassifierAnalyser)
GraphDetailControler(this, lavue)
addView()

III.2 Extensibilité et plugins

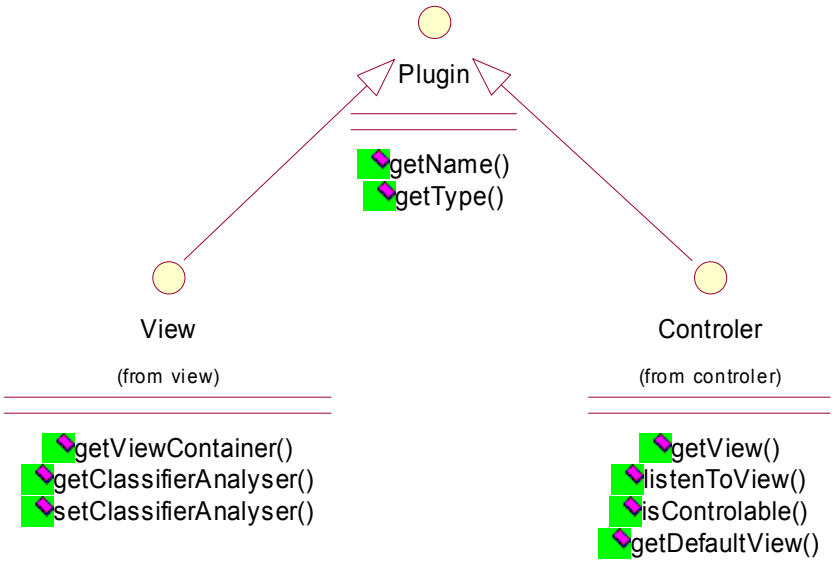
Après avoir établi l'architecture de notre application, il nous a fallu réfléchir aux différentes façons de rendre notre application extensible.

La première chose fut de définir quelles « extensions » pourraient être apportées à l'application. Après avoir discuté avec nos encadrants, il est apparu que l'application devait bien être extensible, mais surtout largement paramétrable. Par exemple, le choix des vues (dans notre cas : graphe d'héritage, graphe de composition/utilisation et vue détaillée) ne devait pas être quelque chose de définitif. Un utilisateur désireux de changer les données apparaissant dans le graphe d'héritage et/ou dans la vue de détail devait pouvoir le faire sans trop de difficultés.

Mais comme nous l'avons expliqué plus haut, notre modèle de données (au sens MVC) est unique, et il s'agit d'une instance de `ClassifierAnalyser`. Comme chaque vue travaille sur un `ClassifierAnalyser`, changer les informations affichées dans une vue n'est pas difficile. Le principal problème était en fait de permettre un développement facile de nouvelles vues (et donc de nouveaux contrôleurs).

Nous avons donc décidé de faire de nos vues et de nos contrôleurs des plugins. Pour cela, nous avons une interface principale, l'interface `Plugin`, que chaque plugin doit implémenter. Puis nous avons des classes abstraites qui correspondent à nos trois vues, ce sont les classes `HeritageView`, `CompositionView` et `DetailView`.

Voici un petit diagramme de classe pour illustrer notre architecture de plugins :



Un utilisateur pourra donc :

Créer un nouveau type d'objet « plugable », en implémentant l'interface `Plugin`

Créer une nouvelle vue ou un nouveau contrôleur, en implémentant l'interface `View` ou `Controler`

Modifier une des vues proposées par défaut en étendant la classe de cette vue.

Les plugins comme leurs noms l'indiquent sont chargeables dynamiquement par l'application. Pour ce faire, il suffit de les placer dans le répertoire nommé `'plugins'` dans le répertoire de base de JavInspector. Le programme construit alors un menu dynamique qui classe les plugins en fonction de leurs types afin que l'utilisateur puisse sélectionner un et un seul plugin pour chaque type. Le chargement dynamique du code du plugin est fait grâce à la classe `PluginLoader` qui utilise la classe `FileClassLoader` que nous avons présentée précédemment.

Nous avons aussi utilisé la généricité pour nos managers. Nous disposons d'une interface `Manager`, puis de classes abstraites `AbstractTopManager`, et `AbstractManager.` La création d'un nouveau manager, pour par exemple, naviguer dans les vues via un menu plutôt que par des onglets, ne pose donc pas de grosses difficultés. Il suffira de créer une nouvelle classe étendant `SplitPaneManager`, puis de surcharger la (ou les) méthode(s) responsable(s) de la création de la partie graphique du manager.

III.3 Présentation de la GUI

Nous allons maintenant étudier plus en détail l'interface graphique de JavInspector. Nous commencerons par une petite description, puis nous regarderons comment nous l'avons implémentée. Enfin, nous parlerons des vues que nous avons choisies d'implémenter.

III.3.1 Description

L'interface graphique de JavInspector est assez classique. Elle est constituée d'un menu principal, qui donne accès à la majorité des actions possibles. Ces mêmes actions sont aussi accessibles via une barre d'outil placée sous le menu. Lorsque l'utilisateur ouvre un classifieur, la fenêtre principale se divise en deux, Du côté droit est affiché le graphe d'héritage du classifieur sélectionné, et du coté gauche sa vue de détail et son graphe de composition/utilisation. Toutes ces vues apparaissent dans des onglets.

Nous avons décidé de grouper la vue de détail et de composition de chaque classifieur dans un même onglet car les informations qu'elles fournissent sont assez complémentaires. Pour naviguer entre les vues (ou lancer de nouvelles vues), l'utilisateur peut utiliser les menus contextuels (pop up menus).

Lorsque qu'une recherche est effectuée, la fenêtre principale (qui contient nos deux groupes de vues) se divise elle-même en deux et les résultats de la recherche lancée sont affichés dans un onglet, ce qui permet de voir les résultats de plusieurs recherches en naviguant simplement dans les onglets. La zone de recherche peut être masquée (ou affichée), grâce à un bouton (elle est par masquée par défaut).

III.3.2 Implémentation

Voici les différentes classes qui constituent notre interface graphique :

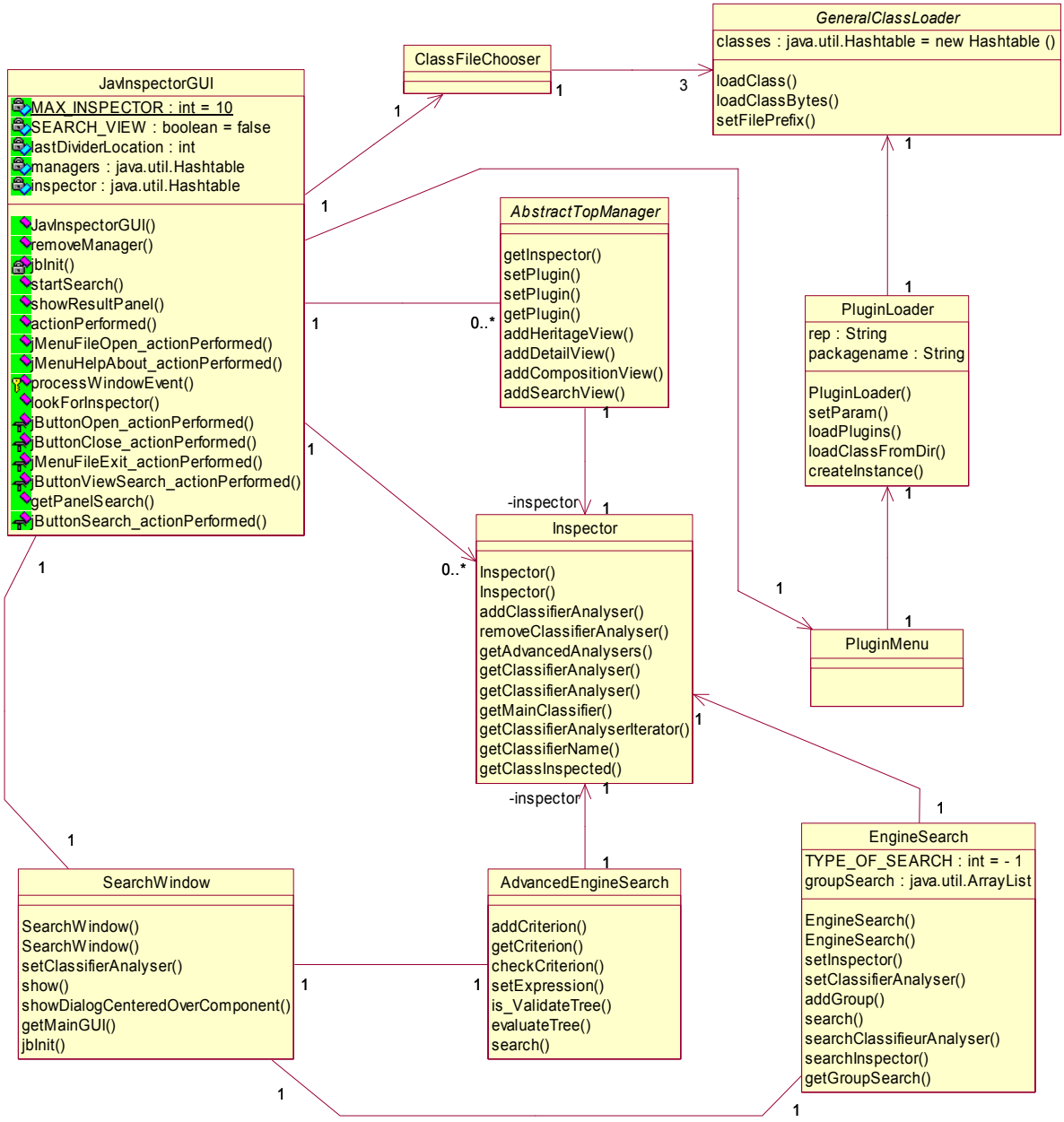
`JavInspectorGUI` **:** il s'agit de la classe principale qui construit l'application. L'instance est unique, c'est elle qui construit les menus, la barre d'outil. De plus elle est responsable des `AstractTopManager`.

`ClassFileChooser` : il s'agit du composant graphique permettant l'ouverture d'un classifieur. C'est une boite de dialogue qui permet d'ouvrir des fichiers .class par différents moyens (par son nom, depuis un jar ou une URL).

`SearchWindow` : un autre composant graphique responsable de la recherche.

`PluginMenu` : c'est un menu spécifique qui se place sur notre `JavInspectorGUI`. Ce menu s'occupe de charger dynamiquement les plugins de l'application.

Et voici le diagramme qui illustre les relations entre ces différentes classes :



III.3.3 Choix des vues sur `ClassifierAnalyser`

Pour réaliser l'interface graphique de JavInspector, nous avons du faire plusieurs choix. L'inspection d'un classifieur Java est assez complexe et les données recueillies peuvent vite devenir très volumineuses. Il a donc fallu identifier les informations qui devaient être affichées. Le JavInspector propose donc trois vues sur les informations collectées sur un classifieur :

La vue de son graphe d'héritage
Une vue de détail du classifieur
Une vue de son graphe de composition/utilisation

Ces vues étant des plugins, l'utilisateur désireux de les changer pourra le faire assez facilement.

III.3.3.1 La vue du graphe d'héritage

C'est la vue qui apparaît à gauche de la fenêtre principale de l'application. Cette vue permet de voir le graphe d'héritage du classifieur inspecté. Si ce classifieur est une classe, cette vue affichera ses classes parentes (jusqu'à la classe mère `java.lang.Object`), ainsi que les interfaces implémentées par chacune de ces classes. Si il s'agit d'une interface, notre vue affichera toutes ses « super-interfaces ». Cette vue permet de naviguer de manière assez rapide parmi les classes « liées » à notre classifieur.

Bien que cette vue ne donne pas accès à un nombre vraiment conséquent de données (pour une classe donnée, seules les interfaces implémentées sont affichées), il suffira à un utilisateur d'étendre la classe de cette vue pour en faire un plugin et décider lui-même des informations qu'il désire afficher.

III.3.3.2 La vue du graphe de composition

Lorsque l'utilisateur a identifié, dans la vue d'héritage, une classe qu'il désire examiner, il peut demander l'affichage de son détail et de son graphe de composition. Comme expliqué ci-dessus, ces deux vues sont groupées, dans notre modèle, cela signifie qu'elle se partage le même manager. Un utilisateur qui décide de fermer la vue de détail fermera par la même occasion la vue de composition. Ce groupement permet de ne pas se perdre parmi les onglets de l'application.

La vue de composition, apparait donc à droite de la fenêtre principale, cette vue est en fait un mélange entre le graphe de composition et celui d'utilisation d'un classifieur. Ce graphe permet en fait de voir toutes les classes « en relation » avec le classifieur inspecté.

Les classes « en relation » avec notre classifieur sont les suivantes :

Les classes des types des variables d'instances et des variables statiques
Les classes des paramètres, des types de retours, des exceptions, des constructeurs et des méthodes du classifieur

Cette vue est intéressante car elle est complémentaire de la vue du graphe d'héritage. Elle permet de pousser plus loin l'inspection, car l'utilisateur peut à nouveau lancer une vue du graphe d'héritage sur n'importe quelle classe du graphe de composition/utilisation et ainsi de suite. Le modèle utilisé par toutes les vues étant identiques (un `ClassifierAnayser`), le passage d'une vue à l'autre et la descente progressive dans les classes n'est pas limitée.

III.3.3.3 La vue de détail

La vue de détail est la vue qui fournie le plus d'informations sur le classifieur. Contrairement aux deux vues précédentes qui ne fournissent que peu d'informations sur le classifieur lui-même, mais mettent plutôt en valeurs ses relations avec les autres classes, la vue de détail ne se préoccupe que du classifieur (sauf dans le cas des méthodes redéfinies).

Les informations affichées sont les suivantes :

Variables d'instances
Variables statiques (ainsi que leurs valeurs qui sont réactulisable)
Constructeurs
Méthodes d'instances et méthodes statiques
Méthodes abstraites
Méthodes des propriétés (méthodes `get*()`, `set*()`)
Méthodes redéfinies (appartenant aux classes parentes)
Classes internes ainsi que les classes anonymes

Comme sur la vue précédente, il est toujours possible de cliquer sur le nom d'une classe pour lancer l'affichage de son graphe d'héritage. L'affichage ou le masquage de certaines informations se fait à l'aide de boutons au sommet de la vue.

La particularité de cette vue est qu'elle peut être rafraîchie, les valeurs des champs statiques pouvant être modifié si la classe est en cours d'utilisation ou si un programme quelconque essaye de les modifier. Cette vue peut de plus être exporté au format XML, il suffit ensuite à l'utilisateur d'utiliser le fichier XSL fourni pour convertir ce fichier au format texte.

III.3.4 Fonctionnalités propres a l'interface graphique

Voici une petite description des fonctionnalités offertes par l'interface graphique de JavInspector :

Inspection de plusieurs classifieurs séparément : on peut lancer une inspection et ouvrir des vues, puis lancer une nouvelle inspection. La première inspection reste toujours accessible via une liste déroulante qui permet de sélectionner l'inspection en cours, en fonction du nom du classifieur ouvert.
Affichage/masquage des résultats des recherches par un clique de bouton.
Stockage des résultats des recherches dans des onglets.
Fermeture des vues et des recherches ouvertes grâce à des icônes placés sur les onglets.
Possibilité d'enficher des vues et des contrôleurs, avec construction dynamique du menu et mise à jour automatique des vues

IV Le moteur recherche

IV.1 Présentation

Dans cette partie, nous allons voir comment nous avons implémenté le moteur de recherche de JavInspector. Nous avons fait le choix de passer du temps à développer le moteur de recherche. En effet, il nous semble important de pouvoir extraire facilement de l'information de nos données. (qui bien souvent sont volumineuses)

Dans ce but, le moteur est capable d'exécuter une recherche sur un `ClassifierAnalyser`, et aussi sur un `Inspector`. On est donc capable de faire une recherche sur un classifieur ou un ensemble de classifieurs.

Pendant le développement du moteur de recherche, plusieurs points importants ont été mis en avant :

Utilisation du design Pattern MVC simplifié (c'est-à-dire sans contrôleur, car dans ce cas il est inutile), afin de séparer efficacement l'interface graphique du moteur de recherche. L'utilisation de MVC permet de changer d'interface graphique sans avoir à réécrire le code du moteur de recherche. On bénéficie donc d'un code réutilisable.

Ecriture structurée de l'API du moteur de recherche avec des classes abstraites (nous avons choisi d'utiliser les classes abstraites pour représenter des modèles de classes à implémenter plutôt que les interfaces ; en effet, notre choix est motivé, et sera justifié un peu plus loin dans la section suivante.). Grâce à une implémentation structurée, le moteur de recherche devient facilement extensible. En effet, l'utilisation de classes abstraites permet d'ajouter ou bien de retirer facilement des critères de recherche au moteur de recherche.

Le module du moteur de recherche se sépare en trois parties distinctes. C'est donc naturellement que notre plan va aller dans ce sens.

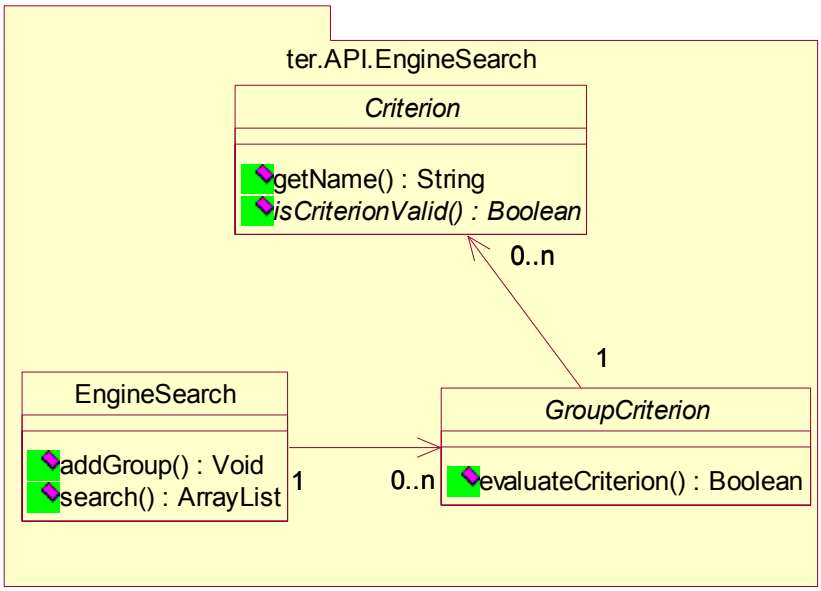
L'API du moteur de recherche de base (module `EngineSearch`).

L'API du moteur de recherche avancée (module `AdvancedEngineSearch`).

Les interfaces graphiques du moteur de recherche basée sur L'API.

IV.2 L'API du moteur de recherche de base

Dans cette partie, nous allons tout d'abord détailler ce qui constitue la base du moteur de recherche. Nous avons perçu le concept d'un moteur de recherche comme suit : « Une recherche est composée de critères. Si un élément que l'on passe en entrée du moteur de recherche est évaluable et qu'il vérifie tous les critères de la recherche, alors il est résultat de la recherche. » C'est sur la base de cette définition, que nous avons imaginé la structure générale de l'API du moteur de recherche. Voici à titre d'illustration, un diagramme de classes simplifié où apparaissent les principaux composants (classes abstraites et la classe principale à instancier)



(Diagramme de classes synthétique de l'API Engine Search).

Comme nous pouvons le voir sur le diagramme ci-dessus, nous avons préféré utiliser des classes abstraites plutôt que des interfaces. Celles-ci ont pour avantage de pouvoir déclarer et initialiser des variables et aussi implémenter des méthodes. Ainsi, dans la classe abstraite `Criterion`, nous avons des méthodes comme `setLogicalValue()`, qui ne seront codées qu'une seule fois. En effet l'action réalisée par cette méthode est la même quelque soit le critère qui sera implémenté à partir de cette classe. De plus, l'utilisation de classes abstraites, oblige un programmeur à implémenter une méthode. Nous n'avons donc trouvé que des avantages à utiliser des classes abstraites.

IV.2.1 Structure du module EngineSearch

Comme nous l'avons précisé plus haut, nous avons apporté une attention particulière à la structure de base de notre moteur de recherche, de manière à ce qu'il soit facilement extensible.

De ce fait, il en découle deux classes abstraites (`Criterion` et `GroupCriterion`) qui vont servir de modèles pour implémenter d'une part des critères de recherche et d'autre part des groupes de critères de recherche.

IV.2.1.1 Criterion ou Critère de recherche

Un critère de recherche constitue la brique de base du moteur de recherche. C'est grâce aux `Criterion` que l'on détermine si un élément est susceptible d'être un résultat de la recherche. Pour cela, la méthode `isCriterionValid()` doit être implémentée par chaque critère, car c'est elle qui détermine si un élément correspond à ce critère. Aussi chaque `Criterion` a besoin d'être identifié par rapport à un autre `Criterion`. Nous avons donc implémenté une fonction `getName()` qui permet de récupérer le nom du `Criterion`. Le fait d'utiliser un nom est pragmatique car il devient facile d'étiqueter de manière visuelle un critère de recherche (par exemple avec des labels sur des boutons). Seulement il faut noter que rien n'empêche à priori d'avoir deux critères qui portent le même nom.

Pour les détails de l'implémentation d'un `Criterion`, nous renvoyons le lecteur à la Javadoc et aux tutoriaux.

IV.2.1.2 GroupCriterion ou Groupes de critères

Les groupes de critères servent à regrouper entre eux des critères de recherche qui portent sur le même « thème ». En effet, un moteur de recherche n'est pas directement composé de tous ses Criterions, et cela pour les deux raisons suivantes.

Tout d'abord, il y a beaucoup de critères dans une recherche, il s'est donc avéré difficile de s'y retrouver.

Deuxièmement, nous avons remarqué que certains critères se regroupent par thème, ce qui permet une hiérarchisation des critères. Cela est fort utile lors de la création d'interfaces graphiques par exemple. De plus, le fait de regrouper certains critères entre eux, permet parfois d'éditer certaines règles au sein même d'un groupe de `Criterion` (exemple : un seul critère à la fois peut être sélectionné dans un groupe). Malheureusement, à cause du temps et des priorités sur l'ensemble du projet, la communication entre critères, par l'intermédiaire du GroupCriterion, n'a pas été implémentée. Cependant, la structure est maintenue en place pour une implémentation ultérieure.

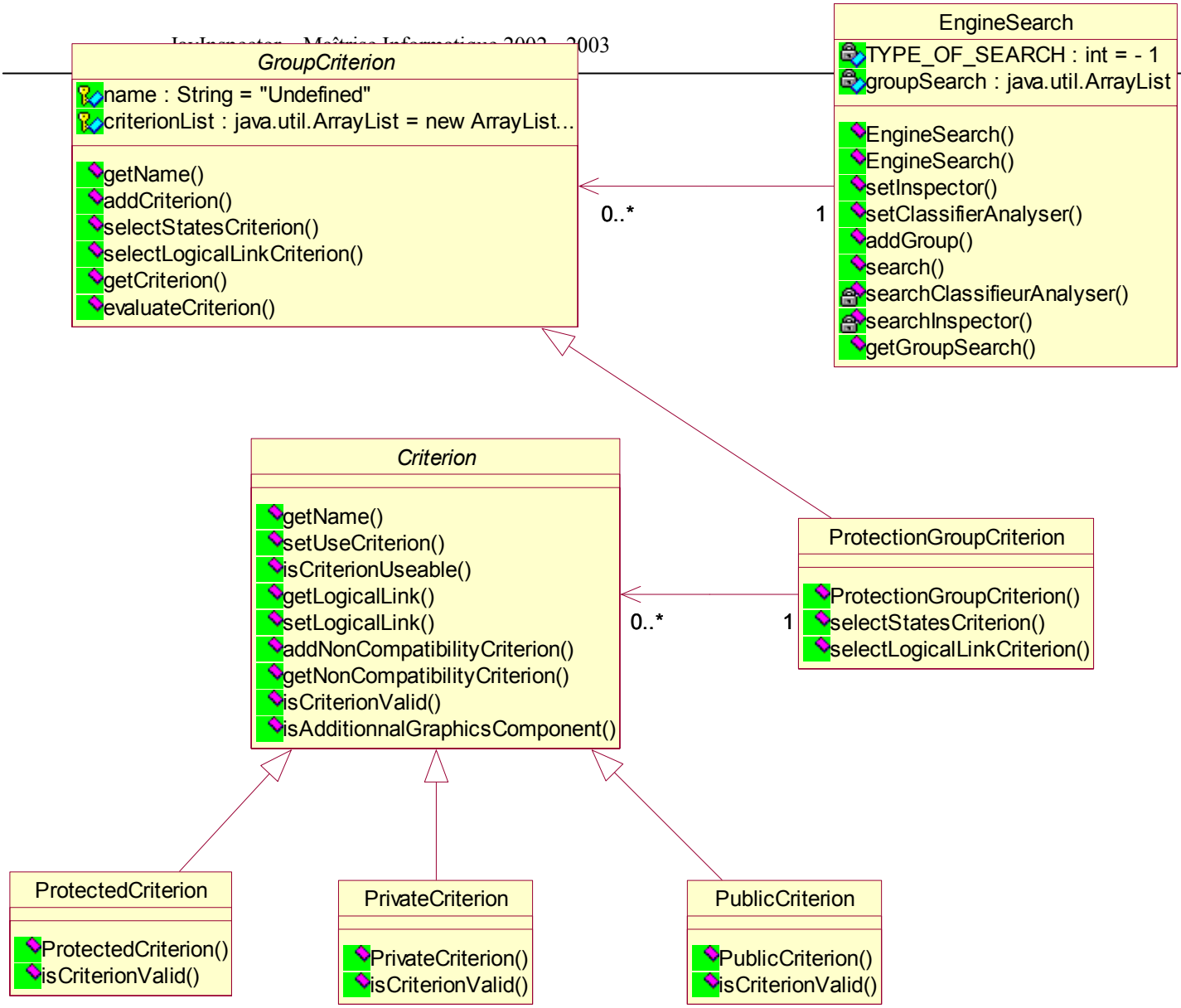
IV.2.2 Fonctionnement de la recherche

Après avoir détaillé la structure du moteur de recherche, nous allons nous intéresser à son fonctionnement. Une fois les `Criterion` et les `GroupCriterion` implémentés, la recherche fonctionne uniquement avec la classe `EngineSearch`. Un `EngineSearch` est fait de manière à être instancié aussi bien avec une instance de `ClassifierAnalyser` qu'avec une instance d'`Inspector`.

Une instance d'`EngineSearch` peut ajouter ou retirer n'importe quel groupe de critères. Ces opérations ne sont possibles que sur les groupes de critères conformément au modèle présenté précédemment.

Avant de lancer une recherche, on peut changer éventuellement l'instance du `ClassifierAnalyser` ou de l'`Inspector` sur lequel on compte effectuer une recherche.

Pour lancer une recherche, il suffit d'appeler la méthode `search()` qui s'occupe de renvoyer la liste de tous les résultats qui correspondent aux critères de la recherche. La recherche s'effectue de la manière suivante : elle prend chaque élément d'un `ClassifierAnalyser` (dans le cas ou l'on fait une recherche de base sur un `ClassifierAnalyser`) et délègue les tests à chacun de ses `GroupCriterion`. Les `GroupCriterion` font de même avec les `Criterion`. Quand les réponses des `Criterion` et des GroupCriterion « remontent », une synthèse des résultats est faite par liaison logique avec l'opérateur booléen AND par défaut.

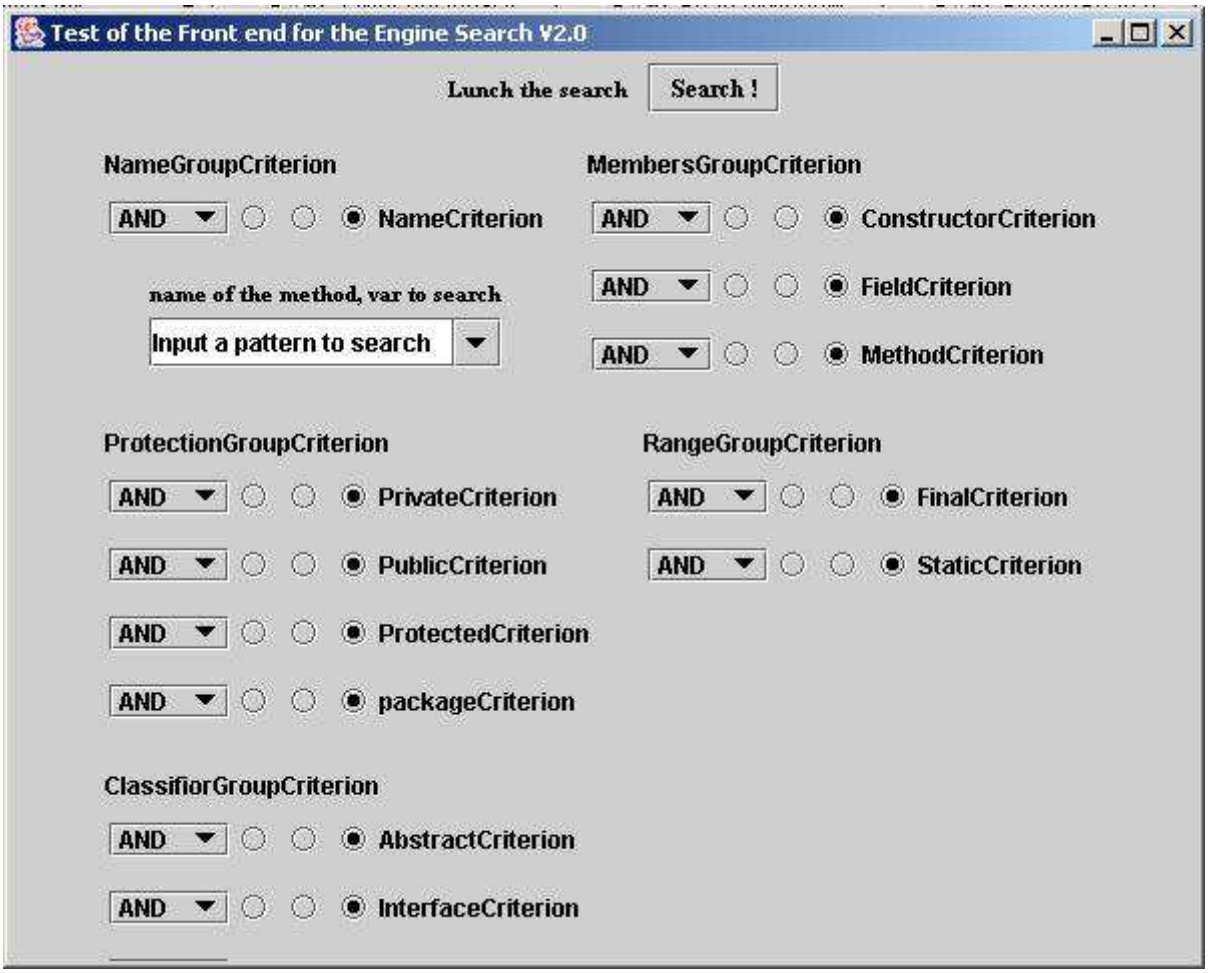


(Un exemple de diagramme qui modélise un moteur de recherche simple avec un groupe de critères.)

IV.2.3 Problème et limite sur la liaison des critères entre eux

Chaque critère implémente la fonction qui vérifie si l'élément correspond au critère. Nous sommes capables de lier les critères entre eux par des opérateurs logiques comme AND et OR, mais l'ordre d'évaluation fige la possibilité de faire des combinaisons entre critères. C'est pour cela qu'il existe un moteur de recherche dit de base (celui qui vient d'être présenté) et un moteur de recherche dit avancé (celui qui va être présenté dans la section suivante).

Voici à titre d'exemple, une interface graphique que nous avons développé au cours du projet, qui utilise le moteur de recherche `EngineSearch.` :



Exemple d'interface graphique pour une recherche.

Cet exemple est une interface graphique qui se génère automatiquement en fonction du moteur de recherche qui lui est passé en paramètre. C'est un bon exemple dans la mesure où il représente exactement de manière graphique le concept de recherche de base. On arrive bien à se représenter quels sont les groupes de critères et les critères qui composent chaque groupe.

De plus, on comprend mieux la notion d'évaluation séquentielle immuable des critères entre eux. Il suffit d'imaginer que chaque critère s'évalue l'un derrière l'autre selon l'opérateur logique qui a été choisi à gauche du critère.

Enfin, en complément sur le `Criterion`, on remarque que pour certains critères de recherche comme le critère de nom avec expression régulière (celui qui retourne vrai si le nom sous forme d'expression régulière correspond au nom de l'élément), on a besoin de saisir de l'information complémentaire (ici la chaîne de caractère qui représente le nom). Pour cela, chaque critère a la possibilité d'implémenter des morceaux d'interfaces graphiques qui permettent de renseigner les champs nécessaires.

IV.3 L'API du moteur de recherche avancée (Advanced Engine Search)

Le but du moteur de recherche avancé est d'offrir la possibilité d'écrire des phrases logiques plus ou moins complexes avec des critères de recherche, service que le moteur de recherche de base ne pouvait pas offrir. Même si le concept de ce moteur est très différent du moteur de recherche de base de part son fonctionnement, il utilise la même structure pour organiser les critères de recherche.

Nous allons voir comment nous avons restructuré ou plutôt enrichi de manière conceptuelle la recherche afin de permettre une plus grande liberté au niveau des requêtes.

Dans la recherche de base, on se contente de paramétrer des critères de recherche, puis on lance une recherche où tous les critères sont évalués séquentiellement. Nous voulons pouvoir paramétrer l'ordre et les priorités dans lesquels les critères sont évalués. Nous avons donc imaginé nous servir du langage des booléens parenthésés.

Voici un exemple simple :

```
Final AND (Public OR Protected).
```

Afin de pouvoir saisir des expressions logiques, nous allons devoir réaliser l'analyse de l'expression. Ce travail revient à réaliser la partie avant d'un compilateur pour notre langage, c'est-à-dire l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique.

L'avantage de notre langage, est qu'il a une grammaire proche de la grammaire ETF. Le seul symbole à gérer en plus est le signe NOT. Après réflexion, ce symbole implique l'ajout de quelques règles dans notre grammaire ETF modifiée. Cependant cela ne pose pas de problèmes majeurs pour la conception d'une grammaire pour laquelle on veut obtenir un parseur automatique d'expressions.

Le travail à réaliser se découpe donc en plusieurs parties distinctes que l'on peut énumérer dans l'ordre suivant :

Analyse lexicale et syntaxique du flux d'entrée et construction d'un arbre syntaxique.
Analyse sémantique de l'arbre syntaxique.
Exécution de la requête par le moteur de recherche avancée.

IV.3.1 Analyse lexicale et syntaxique du flux d'entrée et construction d'un arbre

Afin d'analyser des expressions, nous avons construit une grammaire à partir du langage des expressions booléennes.

Tout d'abord, voici les caractères auxquels l'analyseur n'est pas sensible :

```
SKIP = {" "|"\n"|"\r"|"\r\n"}
```

Ensuite, voici les « tokens » ou mots du langage pour notre grammaire.

```
TOKENS :

LPARENTHESE : "("
RPARENTHESE : ")"
NOT : "!"|"not"|"NOT"
AND : "&"|"and"|"AND"
OR : "|"|"or"|"OR"
MOT : (|"a"-"z"| | |"A"-"Z"|)(|"a"-"z"| | |"A"-"Z"|)*
```

Enfin, voici notre grammaire :

GRAMMAIRE :

```
S  ˘ E <EOF>
AND  ˘ <AND> E
OR  ˘ <OR> E
MOT  ˘ µ
E  ˘ MOT
    | <NOT> <MOT>
    | <NOT> <MOT> AND
    | <NOT> <MOT> OR
    | MOT AND
    | MOT OR
    | NOT
    | <LPARENTHESE> E <RPARENTHESE>
    | <MOT> <LPARENTHESE> E <RPARENTHESE>

LEGENDE : <TOKEN>
```

Quelques remarques sur la grammaire et les « tokens » :

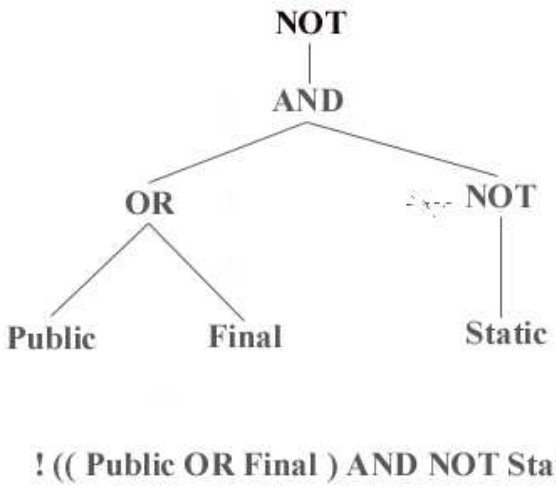
Tout d'abord nous pouvons constater qu'une expression ne peut pas être vide, une requête doit être au minimum un mot.

De plus, un nom de critère ne comporte que des lettres en majuscules ou en minuscules.

Le NOT est prioritaire sur le AND et OR, ainsi : `(! toto AND titi)` sera évalué comme `( ! toto) AND titi`.

Afin de parvenir à générer l'analyseur lexical et syntaxique en Java, nous nous sommes aidés de `javacc` qui s'occupe de générer l'analyseur si on lui fourni une grammaire sur laquelle il peut travailler. Ainsi, grâce à cet outil, nous avons pu générer un analyseur lexico syntaxique. De plus, pendant la phase d'analyse syntaxique, nous générons un arbre syntaxique qui sera utilisé plus tard pour l'analyse sémantique et l'évaluation d'une requête.

Voici un exemple d'arbre valide que l'on peut obtenir après l'analyse syntaxique :



IV.3.2 Analyse sémantique de l'arbre syntaxique

Si l'expression est validée par l'analyse lexico syntaxique, alors il faut effectuer l'analyse sémantique de l'arbre syntaxique (répondre à la question : la requête a-t-elle un sens ?).

En effet si on regarde l'arbre, on s'aperçoit que les nœuds sont des opérateurs logiques de liaison entre critères, et que les feuilles sont des chaînes de caractères qui représentent le nom d'un critère. L'analyse sémantique de l'arbre syntaxique consiste donc uniquement à vérifier que toutes les feuilles ont un nom, qui correspond avec le nom d'un critère que détient la recherche avancée.

Il y a donc deux cas possibles :

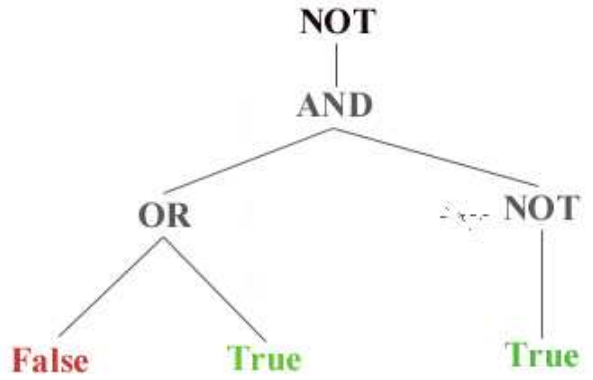
Le nom de la feuille correspond au nom d'un des critères de la recherche avancée, et dans ce cas cette feuille est sémantiquement correcte.

Le nom ne correspond pas et dans ce cas, la feuille est sémantiquement incorrecte, et de ce fait, l'analyseur sémantique renvoie une erreur.

IV.3.3 Exécution de la requête par le moteur de recherche avancée

L'exécution d'une requête n'est possible que si une expression est valide après avoir été compilée. Exécuter une recherche revient simplement à faire un parcours en profondeur de l'arbre syntaxique. Pendant ce parcours, on effectue l'évaluation des feuilles et des nœuds. A la fin du parcours, on obtient donc un résultat booléen qui spécifie si un élément est un résultat de recherche ou non.

Voici, sur la base de l'exemple que nous avons vu plus haut, ce que pourrait être l'évaluation d'une recherche :



! ((False OR True) AND NOT True) = TRUE

IV.4 Les interfaces graphiques construites sur L'API

Dans cette section, nous allons voir quelques composants graphiques que nous avons développés spécialement pour le moteur de recherche.

IV.4.1 Analyse lexicale des mots dans une fenêtre de saisie

Afin de pouvoir offrir la reconnaissance des mots dans une fenêtre graphique, tout comme le font des éditeurs de texte comme `vi` ou `emacs`, nous avons implémenté notre fenêtre de saisie de texte. En effet, nous voulions que notre fenêtre de saisie soit capable de reconnaître certains mots clefs qui sont contenus dans un dictionnaire. Chaque mot du dictionnaire est associé à un style d'écriture. De cette façon, nous sommes capables de changer l'aspect du texte en fonction des mots trouvés dans le dictionnaire.

Voici un exemple de cette fenêtre :

Avant de développer la fenêtre, nous nous sommes demandé s'il fallait faire une analyse complète de la chaîne de caractères saisie comme pour le moteur de recherche avancé. Après réflexion, nous nous sommes aperçus que le problème est différent de celui de l'analyse d'une expression. Il s'agit ici de reconnaître des mots et uniquement cela. Une analyse lexicale suffit donc. Pour mettre à jour le texte de manière régulière, nous avons programmé une `Thread` qui lance l'analyse sur le texte toutes les *n* secondes.

IV.4.2 Construction d'un résultat de recherche graphique

Pour représenter le résultat d'une recherche, nous avons utilisé un arbre dans lequel nous hiérarchisons les données selon leurs `ClassifierAnalyser`, puis selon leur types : méthodes, variables, constructeurs.

Voici un exemple de résultat de recherche :

V Module de sauvegarde.

Nous avons également implémenté un module de sauvegarde pour permettre à l'utilisateur de sauvegarder la vue de détail au format XML.

Le diagramme de classes de ce module est le suivant :

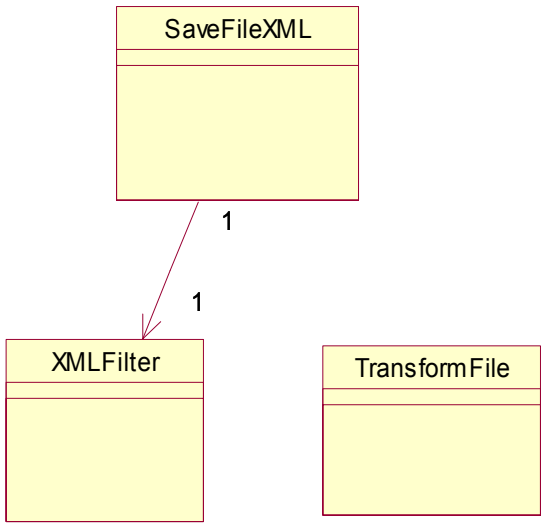


Figure IV.4—1 : Diagramme de classes du module de sauvegarde

Comme on peut le voir sur le diagramme de classes ci-dessus, le module de sauvegarde se compose des classes suivantes :

`SaveFileXML` : cette classe permet de sauvegarder les informations présentes dans la vue de détail au format XML.
`TransformFile` : cette classe permet de transformer un fichier XML en fichier texte grâce à un fichier XSL fournit par l'application.
`XMLFilter` : cette classe fournit un filtre XML pour le choix de sauvegarde d'un fichier.

Le module de sauvegarde permet à l'utilisateur de sauvegarder la vue de détail au format XML. Ensuite celui-ci peut transformer ce fichier XML généré en fichier texte.

La classe `TransformFile` utilise les « parser » et « transformer » fournis par les packages `org.w3c`, `org.xml` et `javax.xml` de la `jdk 1.4.`

Le choix de la sauvegarde au format XML plutôt qu'un autre format réside dans le fait que le fichier pourra ensuite être transformé en un format quelconque par l'utilisateur. Il lui faudra pour cela utiliser un autre fichier XSL que celui fournit par l'application.

CONCLUSION

Nous devions développer une application, JavInspector qui permettrait d'analyser un classifieur Java et d'afficher un maximum d'informations sur celui-ci via une interface graphique agréable et intuitive à utiliser. De plus JavInspector devait être facilement extensible et réutilisable. Nous pensons avoir atteint les objectifs fixés par le sujet. Nous avons attaché beaucoup d'importance à la réalisation d'une interface graphique simple et agréable à utiliser. L'utilisation des « plugins » en fait une application largement paramétrable et extensible. Enfin, le code est soigné et bien documenté.

Le développement de JavInspector nous a beaucoup intéressé et motivé et ce pour diverses raisons. La première fut de savoir que cette application pourrait être diffusée sous Licence GNU LGPL (http://www.fsf.org/licenses/lgpl.html). De plus, le développement des diverses API et de l'interface graphique a certes été contraignant, mais très enrichissant. Bien que la compréhension des « design pattern » nous ait pris beaucoup de temps, leur utilisation s'est avérée très utile pour la réalisation de l'application.

Enfin, le T.E.R nous a permis de mettre en pratique toutes les connaissances acquises durant les deux dernières années d'études passées à la faculté des sciences. En effet, en plus de nos connaissances en Java, nous avons utilisé nos connaissances en analyse, compilation, génie logiciel et UML et avons appris a utiliser `javacc`.

Nous tenons à remercier nos professeurs encadrants pour l'aide qu'ils nous ont apportée tout au long de notre projet.

Bibliographie

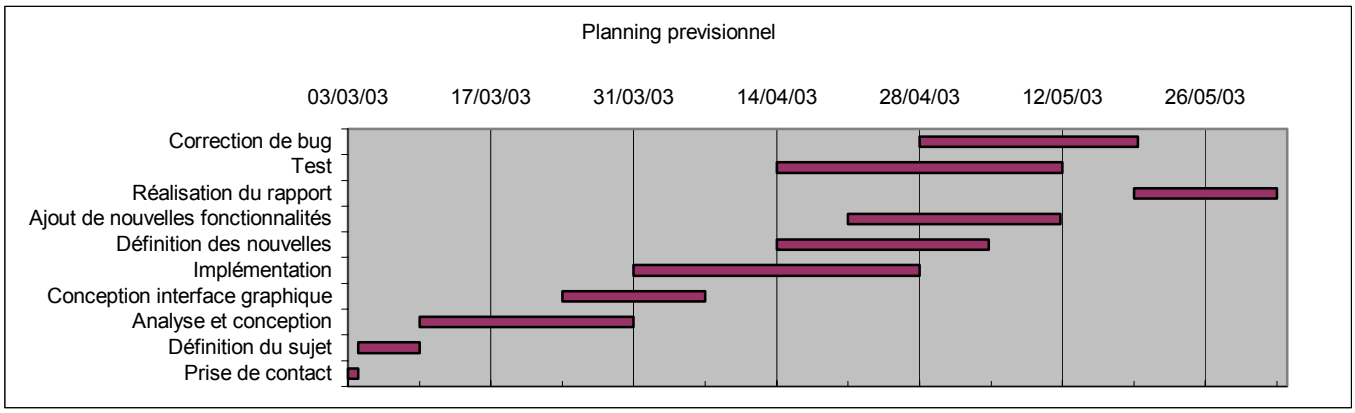
Voici les différents sites utilisés lors de notre projet :

http://forum.java.sun.com/
http://java.sun.com/j2se/1.4.1/docs/api/
http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/
http://java.sun.com/blueprints/patterns/MVC.html
http://www.enode.com/x/markup/tutorial/mvc.html
http://www.wikipedia.org/wiki/Software_design_pattern

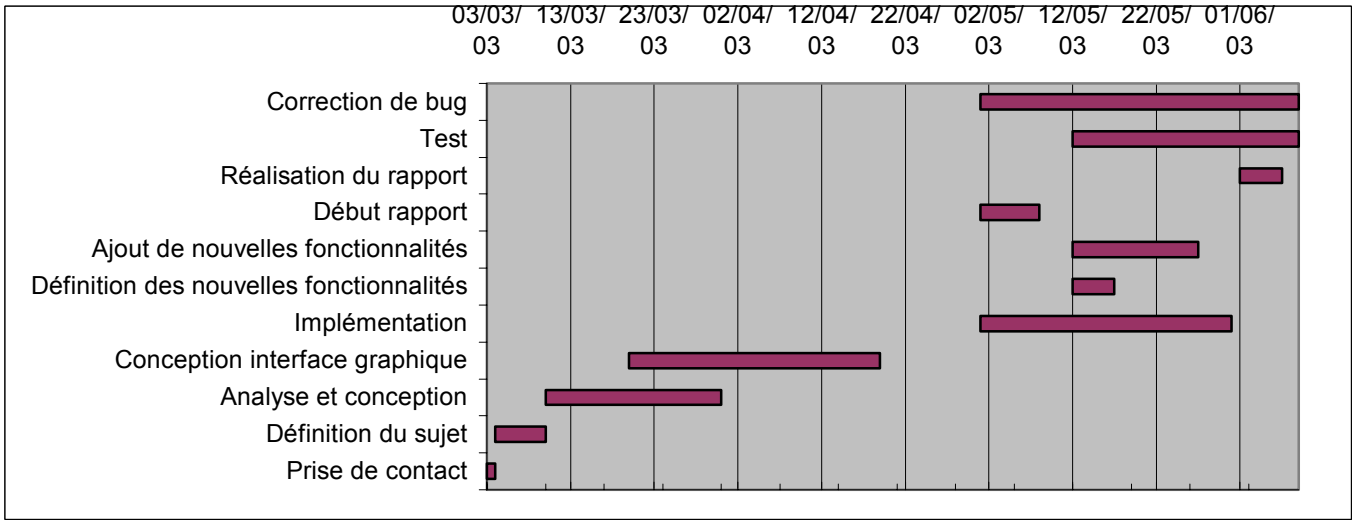
ANNEXES

A Planning

Lors de la première réunion nous avons fourni le planning prévisionnel suivant :



En raison des projets et des partiels du deuxième semestre de la maîtrise ce planning s'est avéré plutôt difficile à respecter. C'est pourquoi nous l'avons modifié à la fin de nos obligations scolaires. Voici la version définitive du planning à laquelle nous nous sommes tenu.



B Installation – Compilation

Pour pouvoir compiler et exécuter JavInspector il faut au préalable avoir installé :

ant
JDK 1.4 de Sun.

JavInspector est fournit sous forme d'archive zip qu'il suffit de décompresser. Cela créera un répertoire JavInspector contenant un script ant :

Pour compiler JavInspector tapez : ant.
Pour exécuter JavInspector tapez : ant run.
Pour générer la « Javadoc » de JavInspector tapez : ant javadoc.
Pour générer une archive jar tapez : ant jar.

Nous fournissons également deux scripts : execute.sh et execute.bat qui compilent et exécutent l'application directement, le premier pour Linux, le second pour Windows.

De plus nous avons pensé à inclure toutes les images utilisées par l'application dans l'archive JAR généré par le script ant, ce qui fait que l'archive jar pourra être utilisée en dehors du répertoire de source de JavInspector.

C Tutoriaux

a) Tutorial de la recherche.

Comment programmer facilement un critère de recherche ?

Programmer un critère de recherche pour un moteur de recherche est très simple.
Il suffit de respecter cet exemple :

```
Public class NouveauCritere extends Criterion{

  public NouveauCritere () {
    this.name = " NouveauCritere ";
  }

  public boolean isCriterionValid(Member m)
      throws SearchEngineException{
    if( isActive == EngineSearchConst.NOT){
      // tester ici si m ne correspond pas à ce critère
    }
    else if( isActive == EngineSearchConst.YES){
      // tester ici si m correspond à ce critère
    }
    else if( isActive == EngineSearchConst.DONTMIND){
      // if i don't mind of the result, I always return true
      // logic !!
      return true;
    }
    else{
     throw new SearchEngineException("isCriterionValid :
Public Criterion : ");
    }
  }

}
```

Exemple d'implémentation d'un Criterion.

Quelques explications :

Le fait d'étendre la classe `Criterion` fournit par défaut un certain nombre de méthodes qui sont déjà implémentées dans cette classe abstraite (elle permettent la liaison avec d'autres classes comme GroupCriterion ou EngineSearch). Il n'y a, comme nous pouvons le voir dans l'exemple, que deux méthodes à implémenter.

La redéfinition de la variable `name` est indispensable, car c'est `name` qui differencie un critère d'un autre. Si ce nom n'est pas surchargé, `name` vaut par défaut `Undefined`.

L'implémentation de **isCriterionValid(Member**) est importante, car c'est cette methode qui décide si un Member (classe mère de Method, Class, Field) correspond à ce critère.

Comment ajouter un critère de recherche à un moteur de recherche ?

Ajouter un critère dans un moteur de recherche est vraiment très simple.
Si le moteur de recherche utilisé est EngineSearch, il y a deux solutions :

Premièrement, si le critère de recherche peut s'intégrer dans un groupe de critères, il suffit de choisir la classe qui correspond au groupe, et de lui ajouter dans son constructeur une ligne qui ajoute le `Criterion` (ainsi c'est fait de manière définitive), ou alors, se servir de la méthode `addCriterion(Criterion c)` de GroupCriterion pour ajouter de manière dynamique un critère au group de Criterion.

```
public class monGroupCriterion extends GroupCriterion {

  public monGroupCriterion(){
    name = "monGroupCriterion";

    // add here the criterion you want
    criterionList.add(new PrivateCriterion());
    criterionList.add(new PublicCriterion());
    criterionList.add(new ProtectedCriterion());
    criterionList.add(new packageCriterion());
...
  }
```

Exemple d'implémentation d'un GroupCriterion.

Cet exemple illustre bien comment ajouter en « dur » un `Criterion` à un groupe de critères. De plus, cela montre tout ce qu'il faut faire pour implémenter un groupe de critères.

Si le moteur de recherche est avancé, il suffit d'appeler la méthode `addCriterion(Criterion c)` pour ajouter un critère au moteur de recherche.

b) Tutorial d'implémentation de plugins

Le but de ce tutorial est de fournir les informations à un programmeur qui voudrait étendre le JavInspector en créant de nouveaux plugins.

Notre application s'occupe de six types de plugins différents :

Vue et contrôleur pour l'héritage
Vue et contrôleur pour le détail
Vue et contrôleur pour la composition

Pour créer un plugin vous devez implémenter la classe abstraite qui correspond au type du plugin que vous voulez programmer.

Par exemple :

```
public class TreeDetailView extends ter.SDK.view.DetailView {

}
```

Pour plus de détails sur le rôle des fonctions à implémenter, vous devez consulter la Javadoc fournie avec le programme.

Cependant vous devez faire attention à fournir dans tous les cas le constructeur par défaut :

```
public TreeDetailView() {

}
```

Ainsi que selon le type (vue ou contrôleur) les constructeurs suivant :

Pour une vue :

```
public TreeDetailView(ter.API.ClassifierAnalyser ) {

}
```

Pour un contôleur :

```
public TreeDetailControler(ter.SDK.manager.Manager ,
                                ter.SDK.view.View) {

}
```

Attention dans les deux cas le type des paramètres du constructeur doit être exactement celui décrit précédemment, et non le type d'une classe qui l'étend.

D Manuel de l'utilisateur de JavInspector

Pour le détail des commandes, se référer la capture d'écran à la fin du manuel.

a) Ouverture d'un fichier `'.class'`

Il y a deux façons d'ouvrir un fichier .class, soit en cliquant sur le bouton open soit dans le menu : File ˘ Open a classifier. A ce moment, une fenêtre s'ouvre et offre à l'utilisateur plusieurs options :

Soit il coche la case « Name » et il peut ouvrir un fichier du CLASSPATH ou une classe de la JDK en tapant directement son nom sans le suffixe .class dans la zone de saisie « Name ».
Soit il coche la case « File » a ce moment la il peut ouvrir un fichier .class en appuyant sur le bouton « Find … ».
Soit il coche la case « JAR », alors il doit taper le nom de la classe dans la zone de saisie « Name » puis chercher l'archive JAR en appuyant sur le bouton « Find… ».

b) Navigation dans l'interface graphique

i) Navigation dans le graphe d'héritage

Une fois un classifieur ouvert, son graphe d'héritage apparaît dans la partie gauche de l'écran. L'utilisateur peut double cliquer sur la racine du graphe pour le voir en entier. Une fois ouvert, le graphe d'héritage offre à l'utilisateur plusieurs options sur n'importe quel classifieur lui appartenant. Pour activer ses options, il lui suffit de faire un clique droit de souris sur le nom d'un classifieur quelconque ce qui fait apparaître un menu contextuel offrant les possibilités suivantes :

« Show DAC view »: ouverture de la vue de détail et du graphe de composition/utilisation du classifieur dans deux onglets différents à droite de l'application.
« Search in classifier » : recherche dans le classifieur selectionné.

ii) Navigation dans la vue de détail.

Une fois la vue de détail ouverte, l'utilisateur peut voir et cacher chacun de ses éléments en cochant/décochant les cases situées en haut de la vue. Il peut également « double cliquer » sur n'importe quel type de classe, ce qui affichera son graphe d'héritage à gauche.

De plus, l'utilisateur peut à tout moment effectuer un clique droit sur la vue ce qui fait apparaître un menu contextuel offrant les options suivantes :

« Save as… » : sauvegarde de la vue au format XML
« Refresh view » : rafraîchir la vue.
« Search » : effectuer une recherche dans la vue.iii) Navigation dans le graphe de composition.

Il est possible de faire apparaître/disparaître chaque partie du graphe de composition en cliquant sur le bouton de la partie désirée.

c) Utiliser la recherche

Il est possible d'effectuer des recherches sur un classifieur ou sur un inspecteur. Pour cela, il suffit de cliquer sur le bouton de recherche situé en haut à droite dans la barre d'outils. On peut y faire une recherche simple (saisie de la chaîne de caractère à chercher), ou alors une recherche avancée. La recherche avancée demande une requête sous forme d'expression logique.(ex : Public OR Private). Les opérateurs permis sont : AND OR NOT ( ).

Voici un exemple plus complexe : !((Final OR !static) AND (Public OR Private)).Les résultats de la recherche sont ensuite affichés dans une fenêtre se situant en bas de l'application. Pour faire apparaître ou masquer ces résultats, il suffit de cliquer sur le second bouton de recherche.

d) Capture d'écran

Voici une capture d'écran montrant les différents boutons et leurs actions ainsi que les différentes vues proposées par l'application.