Janx: un navigateur de classes - projet de Programmation Java, Maîtrise Informatique, UMLV

Cécile MICHELOU et Daniele RAFFO

25 janvier 2001

Table des matières

Chapitre 1

Introduction

Le but de ce projet est de réaliser un navigateur de classes Java compilées, c'est à dire un logiciel capable de prendre en charge des fichiers .class et d'en afficher les membres qui y sont contenus, avec toutes les informations concernant leur nom, visibilité, modificateurs, classe de déclaration, etc. . Un tel mécanisme est appelé *introspection* ou *réflexion*.
Nous avons donc pour ce but developpé un programme que nous avons appelé “Janx”.
Le tout est accessible à partir d'une interface graphique.

Note : dans ce rapport, nous utilisons le terme *membre* pour indiquer un champ ou une méthode. Un *champ* est une variable (de type int, float, boolean, Object...). Ces dénominations sont paralléles à l'usage fait dans le API de Java, où on trouve des objets Member, Field et Method.

Chapitre 2

Utilisation du programme

Le programme est démarré au moyen de la commande `java -jar Janx.jar`.
Dans la suite nous décrirons les différentes composantes de l'application.

2.1 L'arborescence des classes (fenêtre à gauche)

La fenêtre à gauche est dediée à la liste des classes dont on effectuera la réflexion.
Cette liste est décrite dans ce que nous avons appelé un fichier *classlist*.

Un fichier classlist est un fichier ASCII qui contient la liste des répertoires ou des fichiers Jar (*Java Archive*) où se trouvent les classes à charger, en raison d'un nom par ligne (terminée par un caractére de newline). Les répertoires et les Jar doivent être identifiés par leur path absolu. Il n'est pas nécessaire que le fichier classlist ait un nom ou une extension particuliére.

Un exemple de fichier classlist est fourni avec notre programme avec le nom `monfichier`. Ceci est son contenu :

```
/home/main01/draffo/PROGRAMMATION_OBJET
/home/main01/draffo/PROGRAMMATION_OBJET/PROJET
/home/main01/draffo/PROGRAMMATION_OBJET/TD5
/home/main01/cmichelo/MAITRISE/JAVA/classes
/home/main01/draffo/PROGRAMMATION_OBJET/PROJET/cecile.jar
```

Une fois le fichier classlist chargé, la fenêtre contiendra donc l'arborescence des classes. La racine est le nom du classlist ; chaque noeud correspond à un répertoire ou à un fichier Jar du classlist, et contient la liste de tous les fichiers .class présents à l'intérieur. Si le programme a trouvé des erreurs dans les lignes du classlist (répertoire ou Jar non existant ou illisible), ils sont affichés sur la sortie standard. Si un répertoire ne contient pas de fichiers .class, le noeud correspondant est montré dans l'arborescence mais il contient un drapeau spécial.

Important : il faut que la variable de systéme $CLASSPATH (qui définie le path des fichiers pour Java) inclut les répertoires suivants :
- le répertoire où se trouve le fichier classlist
- les répertoires listés dans le fichier classlist lui-même
- le path des fichiers Jar listés dans le fichier classlist lui-même

Faute de quoi, il sera possible de charger seulement le fichier classlist et d'introspecter seulement les classes, du répertoire où se trouve Janx.

Au démarrage, aprés l'affichage d'un écran d'information avec les noms des auteurs, le programme charge le fichier classlist spécifié dans le fichier de configuration `Janx.cfg`.
Si `Janx.cfg` n'existe pas, l'arborescence sera vide et le programme affichera au départ un message qui invite l'utilisateur à charger un fichier classlist au moyen du menu.
Il est possible à tout moment de charger un autre fichier classlist, en sélectionnant l'option **Load classlist file** d'aprés le menu.

Pour effectuer l'introspection d'une classe, on clique dessus.

2.2 La table des membres (fenêtre en bas à droite)

Lorsqu'on clique sur une classe dans l'arborescence, elle est donc introspectée. Dans la fenêtre en bas à droite apparaît alors, aprés quelques instants, une table contenant la liste des champs et des méthodes auquels elle a accés.

Chaque ligne correspond à un membre. Les informations suivantes (une pour chaque colonne, dans l'ordre) sont affichées :
- **Member** : champ ou méthode
- **Visibility** : visibilité du membre (private, protected, public ou sans modificateur)
- **More modifiers** : des autres modificateurs qui peuvent affecter le membre (static, final, synchronized...)
- **Type** : le type (pour les champs seulement)
- **Return type** : le type de retour (pour les méthodes seulement)
- **Name** : le nom du membre
- **Arguments** : les types des arguments d'appel, dans l'ordre, separés par une virgule (pour les méthodes seulement)

Les champs sont montrés toujours avant les méthodes, et les membres statiques avant ceux qui ne le sont pas.
Il est possible de trier les membres par nom ou par visibilité. Ceci est effectué en cliquant sur le bouton d'en-tête de la colonne correspondante au tri desiré.

2.3 La hiérarchie des classes (fenêtre en haut à droite)

Le graphe de la hiérarchie de la classe choisie apparaît au même instant que la table, dans la fenêtre droite supérieure. La classe introspectée est montrée en rouge. En dessous d'elle sont montrées en jaune ses super-classes eventuelles et, sur le côté, les interfaces eventuelles qu'elle implante, toujours de couleur jaune.

En cliquant sur un membre dans la table, la classe où il a eté déclaré se teint d'une couleur bleue ; si le membre a eté redéfini dans certaines classes, ces dernières deviendront vertes claires.

2.4 Le menu

Le menu permet de faire un choix parmi ces options :
Load classlist file : charge un fichier classlist.
Save configuration : sauvegarde le nom du fichier classlist actuel (celui dont le nom apparaît dans la racine de l'arborescence) dans la configuration. Au démarrage de la session suivante, ce classlist sera automatiquement chargé.
About : montre un écran d'information sur le programme.
Quit : sortie du programme.

Chapitre 3

Implémentation

Le projet, nommé "Janx", a eté écrit en Java 1.2.2 sous Linux, avec interface graphique Swing.
Une documentation en format javadoc a eté aussi engendrée et est accessible dans
`home/main01/draffo/PROGRAMMATION_OBJET/PROJET/Janx-javadocs`.
Ces javadocs expliquent en détail le fonctionnement des classes et des méthodes que nous avons developpé, toutefois, ici nous donnons des éclairements sur certaines opérations du programme.

3.1 Chargement des classes

Dans la classe ChargeurClasse, nous avons redéfini un chargeur de classe (*classloader*) personnel. Il est utilisé à l'intérieur de la classe ExaminateurClasse pour le chargement des fichiers .class.

3.2 Chargement d'une classe à partir d'un fichier Jar

Pour le chargement d'une classe contenue dans un fichier Java Archive, la façon de procéder est différente. D'abord on explore l'intérieur du fichier Jar pour en récupérer la classe ; ensuite, il faut décompresser cette classe pour en charger le *bytecode*.

3.3 Récuperation des membres

Pour la récuperation de tous les membres accessibles d'aprés la classe sélectionnée, il n'a pas eté possible d'utiliser getFields() et getMethods() de java.lang.Class, car ils ne retournent que les membres publiques d'une classe. On a donc utilisé getDeclaredFields() et getDeclaredMethods(), qui retournent les membres (de toutes visibilités) explicitement déclarés dans la classe. On les appelle sur la classe choisie et récursivement sur toutes ses super-classes, en mémorisant les résultats dans deux listes globales (une pour les champs et l'autre pour les méthodes) qui se remplissent au fur et à mesure que les appels de fonction se déroulent.
Ceci pose deux problèmes à considérer :

1. les contraintes de visibilité
2. la possible redéfinition d'un membre dans une classe, quand il avait dejà eté déclaré à l'interieur d'une super-classe (*masquage*)

Pour résoudre ces deux problèmes, on a donc developpé ces deux test :

1. Test de visibilité : le membre est accepté s'il est protegé (car il est alors visible dans toute sous-classe) ou publique, et il est refusé s'il est privé. S'il n'a pas de modificateur associé, on n'accepte le membre que s'il appartient au même paquetage de la classe reflectée. (Ceci a eté modifié : cf. chapitre "Bogues").
2. Test de masquage : on parcours la liste globale et on vérifie si le membre a la même signature (si c'est une méthode), ou a le même nom (si c'est un champ), d'un des membres déjà présents dans la

liste globale ; s'il l'est, alors ce membre a eté masqué et il faut le refuser.

Rappelons que, à tout moment, quand on examine les membres déclares d'une classe C, la liste globale contient tous les membres déclarés dans les sous-classes de C.

Aprés avoir obtenu les membres d'une (super-)classe au moyen de getDeclaredFields() et getDeclaredMethods(), on applique donc les deux test à chaque membre ; si le membre passe les deux test, il est ajouté à la liste globale, qui va donc à la fin contenir tous les membres accessibles à partir de la classe reflectée.
Les test se déroulent selon ces critéres :

- Pour la classe dont on est ent train de faire l'introspection :
  Trivial. Aucun test de visibilité (car tous les membres sont visibles à elle-même), aucun test de masquage (cela n'aurait pas de sens).
- Pour sa super-classe immediate :
  On effectue le test de visibilité, mais pas le test de masquage.
  A propos de ce dernier point, nous rappelons que les membres d'une classe masqués dans une sous-classe, sont toujours visibles à partir de la sous-classe au moyen du mot-clé `super`, et ils restent donc accessibles. Ceci n'est valable que pour un seul niveau d'héritage, `super.super` étant illégal.
- Pour toutes les autres super-classes :
  On effectue les tests de visibilité et de masquage.

Une remarque importante : la visibilité d'un membre est limitée par la visibilité de sa classe déclarante. Un algorithme de calcul de la visibilité réelle demandant une complexité trop grande (on ne peut pas utiliser getModifiers() pour les membres, donc il faudrait garder en mémoire une autre structure de données pour la mémorisation de leur visibilités réelles), cela n'a pas eté pris en compte dans notre projet.

3.4 Recherche des classes de déclaration / masquage

Lorsqu'on sélectionne un membre dans la table, on obtient les valeurs des différents cellules, sous la forme de chaînes de caractéres (String). (Ils s'agit des résultats des appels à getName(), getReturnType.getName(), etc. sur l'objet Field ou Method.) On n'a donc pas pu obtenir un objet de type Member pour lui appliquer getDeclaringClass(), comme on pensait de faire. On a donc utilisé la tactique suivante.

Pendant la récupération des membres (voir section précédente), on mémorise dans un tableau (Vector) tous les membres déclarés par la classe actuelle, sans en éliminer aucun, en les stockant sous forme de chaînes de caractéres, avec le nom de la classe comme premier élément ; ceci est répeté pour toutes les super-classes, et tous les tableaux ainsi obtenus sont regroupés dans un autre tableau (Vector), qui réprésente donc le schéma de déclaration des membres.
Quand une ligne de la table est sélectionnée, on reconstruit, d'après les valeurs des cellules correspondantes, le nom du champ ou la signature de la méthode, selon le même critére utilisé auparavant. Ensuite on compare cette donnée avec le schéma de déclaration, et on en déduit une liste (ordonnée et pas vide) de classes. Toutes ces classes déclarent le membre sélectionné dans la table ; la dernière classe de la liste est la classe qui effectue la déclaration initiale, toutes les autres effectuent un masquage.

3.5 Affichage des membres

Le tri des membres dans la JTable a lieu de la visibilité la moins restreinte vers la visibilité la plus restreinte, avec pour chaque visibilité un tri par ordre alphabétique. Pour ce tri, on a implémenté la méthode compareTo() d'un comparateur (Comparator).

Pour créér la table, on a utilisé un modéle qui contient les données. Ce modèle s'appelle TableModel.

3.6 Affichage de la hiérarchie

Pour le graphe d'affichage de la hiérarchie, on a utilisé un programme issu des démonstrations des API Java.

3.7 Sauvegarde de la configuration

Le nom du fichier classlist à charger au demarrage est memorisé dans un fichier de propriétés (classe Properties), nommé `Janx.cfg`. Les méthodes qui s'occupent de sa sauvegarde et de sa lecture sont Navigateur.sauvegardeProprietes() et Navigateur.chargeProprietes().

Chapitre 4

Liste des classes

4.1 Navigateur

Interface graphique et flot d'execution du programme.

4.2 LecteurFichier

Examine un fichier classlist et y récupere les noms des classes.

4.3 ChargeurClasse

Notre chargeur de classe, utilisé pour le chargement des classes à introspecter.

4.4 ExaminateurClasse

Classe qui s'occupe de la réflexion. Elle récupére les membres et découvre la hiérarchie d'une classe. Son constructeur appelle le chargeur de classe personnel redéfini.

4.5 TableMethodField

Construit et gére la table des membres.

4.6 FiltreTriModel

Gére le tri des membres dans la JTable.

4.6.1 Ligne

Définit la fonction de tri compareTo().

4.7 Graph

Graphe de la hiérarchie des classes.

4.7.1 Node

Etiquette qui répresente une classe du graphe.

4.7.2 Edge

Lien entre les classes (Node).

4.7.3 GraphPanel

Gére le dessin.

4.8 UtilMembres

Contient des méthodes statiques utiles pour le travail sur les membres introspectés (visibilité, egalité, répresentation sous forme de chaîne de caractéres...)

Chapitre 5

Format du projet

Le projet est composé des fichiers suivants. Les fichiers .class et .java sont fourni dans un Java Archive nommé `Janx.jar`.

ChargeurClasse.java
ExaminateurClasse.java
FiltreTriModel.java
Graph.java
LecteurFichier.java
Navigateur.java
TableMethodField.java
UtilMembres.java
ChargeurClasse.class
Edge.class
ExaminateurClasse.class
FiltreTriModel$1.class
FiltreTriModel$Ligne.class
FiltreTriModel.class
Graph.class
GraphPanel.class
LecteurFichier.class
Navigateur$1.class
Navigateur$2.class
Navigateur.class
Node.class
TableMethodField$1.class
TableMethodField.class
UtilMembres.class
monfichier (un example de fichier classlist)
Janx.cfg (le fichier de configuration de Janx)
image.gif (le logo montré dans l'écran d'informations)
Janx-rapport.ps (ce rapport)

Chapitre 6

Bogues

En ce qui concerne l'affichage de la table des membres, on a deux bogues pour lesquels il n'a pas eté possible de trouver la cause et que nous reportons ici.

Dans la méthode UtilMember.estVisible(), il a eté nécessaire de désactiver le test (qui a donc eté mis en commentaire) qui vérifie si un membre sans modificateur de visibilité appartient au même paquetage d'une classe. En fait, quand ce test était activé, la table des membres ne donnait pas des résultats corrects : par exemple, elle n'affichait pas certains champs privés et sans modificateurs, et certaines méthodes (dont la méthode `public static main()`), de `Navigateur.class`.

Si dans la table des membres apparaît deux champs de même nom ou deux méthodes de même signature, c'est que le membre a eté déclaré dans une super-classe de la classe introspectée, et redéfini dans la classe introspectée elle-même (comme nous avons expliqué précédemment). Ceci est normal.
Cependant, rarement il arrive que soient montrés deux membres égaux qui ont eté déclarés et redéfinis seulement dans les super-classes. Dans ce cas là, normalement, la table serait censée afficher une seule version du membre : celle masquée dans la plus proche super-classe de la classe reflectée.

Pour ce qui concerne la gestion des fichiers Jar, on en a géré l'ouverture et l'exploration des classes contenues. Cependant, le systéme ne gére pas le cas d'interdépendance des classes, c'est à dire quand la classe chargée fait appel à d'autres classes du fichier Jar.

Chapitre 7

Conclusion

Pendant le développement de ce projet, nous avons pu aperçevoir la puissance du mécanisme de la réfléxion en Java.

Son utilisation permet de récupérer dynamiquement une grande quantité d'informations à partir du *bytecode* pur, sans avoir besoin des sources. Ces informations peuvent être utilisées pour écrire du code qui va au delà de certaines contraintes de sécurité (visibilité, etc.). En effet, une introspection de méthodes permettrait aussi d'avoir une sorte de "pointeur de fonctions" qui sont présents dans plusieurs langages (C, C++) mais absents, pour ces questions de sécurité, en Java.