Système de recherche de méthodes *Java* basé sur leur signature

Nicolas Bonnel* **et Gurvan Le Guernic***,†

* IRISA / Université de Rennes 1
Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
{nicolas.bonnel,gurvan.le_guernic}@irisa.fr

† Kansas State University, Manhattan, KS 66506, USA

Résumé : L'objectif de cet article est de proposer une démarche permettant de mettre en place un moteur de recherche de méthodes au sein d'un langage de programmation. Un tel outil s'avère particulièrement utile aux développeurs. Des solutions ont déjà été proposées mais elles sont pour la plupart basées sur une recherche textuelle, c'est-à-dire uniquement basées sur le contenu textuel de la description des différentes méthodes. Nous proposons dans cet article une nouvelle approche basée sur la signature des méthodes. Le langage utilisé tout au long de cet article est le langage *Java*.

Mots-clés : système d'information, génie logiciel, réutilisation, recherche de méthodes, *Java*.

1 INTRODUCTION

Les langages de programmation, tels que *Java* qui est le langage utilisé dans cet article, offrent aux développeurs un nombre important de bibliothèques afin de faciliter leur tâche. Ainsi lorsqu'un développeur veut stocker des données dans une structure, il n'a pas besoin de coder cette dernière. Il lui suffit d'utiliser une des structures déjà codées dans une bibliothèque du langage. Par exemple, dans le cas de *Java*, il dispose du paquetage[1] *java.util* qui contient, entre autres, des classes pour les listes, les tableaux, les vecteurs ou encore les tables de hachage. Alors, le développeur n'a plus qu'à déterminer la structure qui convient le mieux à son problème, et à trouver les méthodes accomplissant les actions qu'il désire. Cependant, ces bibliothèques sont énormes. Elles contiennent un très grand nombre de classes et un nombre encore plus grand de méthodes. De ce fait, il est parfois difficile de déterminer quelle est la méthode qui accomplit l'action que l'on souhaite réaliser. Il existe alors des outils pour aider le programmeur dans sa recherche. À de rares exceptions près, ces outils se basent uniquement sur la recherche de mots-clés. De plus, tous ces outils présentent leurs résultats sous la forme d'une liste. Par exemple, *Eclipse*[2] inclut un module de recherche par mots-clés. Plus évolué, l'outil *Method Finder* de *Squeak*[3] permet d'effectuer, en plus de la recherche par mots-clés, une recherche par l'exemple. Dans ce type de recherche, l'utilisateur fournit un exemple d'utilisation de ce qu'il cherche. Cet exemple est composé des valeurs des paramètres de la méthode recherchée et de celle du résultat escompté.

Les travaux présentés dans cet article abordent le problème de la recherche de méthodes sous un angle différent. Tout d'abord, la recherche se base non pas sur des mots-clés ou des exemples d'utilisation, mais sur la signature[4] de la méthode [Rittri 1991, Runciman 1991]. Cette approche correspond à la démarche empruntée, dans un premier temps, par l'utilisateur lors de la recherche d'une méthode dans une API[5]. Enfin, l'application de techniques du domaine de la visualisation de l'information permet de représenter les résultats obtenus sur un plan. Cette représentation permet à l'utilisateur de prendre connaissance rapidement de la pertinence des résultats, ainsi que de leurs similarités.

L'article est organisé de la façon suivante. La prochaine section présente l'approche utilisée d'un point de vue général. Ensuite, la représentation des données est abordée dans la section 3, et la section 4 traite du mécanisme sous-jacent à la recherche des méthodes. Puis, la section 5 s'intéresse à la visualisation des résultats de recherche. Afin de valider notre approche, un prototype a été développé. Il permet de rechercher des méthodes dans un sous-ensemble des classes du langage *Java*. Il est présenté succinctement en section 6 avec une interprétation des résultats. Enfin, les deux dernières sections sont respectivement consacrées aux travaux similaires puis à un bilan accompagné de perspectives.

2 PRÉSENTATION DE L'APPROCHE

L'approche proposée dans cet article diffère des approches habituelles à la fois par le mécanisme utilisé pour

[1] Traduction francophone du terme *package*.

[2] `http://www.eclipse.org`

[3] `http://www.squeak.org`

[4] Dans notre cas, la signature est composée des types des paramètres et du résultat de la méthode.

[5] *Application Programming Interface* — Bibliothèque de fonctions destinées à être utilisées par les programmeurs dans leurs applications.

Actes de la 4e Manifestation des Jeunes Chercheurs en Sciences et Technologies de l'Information et de la Communication (MajecSTIC'06), Lorient, France, 22-24 novembre 2006.

la recherche de résultats, ainsi que par la présentation de ces résultats à l'utilisateur.

2.1 Principe de recherche

Habituellement, la recherche de méthodes basée sur les signatures fournit des résultats répondant « exactement » à la requête de l'utilisateur [Rittri 1991, Runciman 1991, Di Cosmo 1993]. Le terme « exactement » est employé dans le sens où les résultats sont directement applicables à la requête. Aucune modification, même légère, des paramètres n'est à effectuer — comme par exemple, le passage d'une valeur de type entier (`int`) à une valeur de type flottant (`float`). Or, l'utilisateur peut parfois se satisfaire de méthodes ne répondant pas « exactement » à sa requête. Par exemple, s'il cherche à calculer la racine carrée d'un entier et que cette méthode n'est pas implémentée pour les entiers, il peut utiliser sans difficulté une méthode calculant la racine carrée d'un nombre flottant. Malheureusement, la plupart des outils actuels ne rendent pas en réponse des méthodes travaillant sur les flottants si la requête concerne des entiers.

Le mécanisme que nous proposons ne se limite pas aux réponses « exactes ». Les résultats retournés sont des méthodes considérées comme « proches » de la requête. Cette notion de proximité se base sur un calcul de distances[6] entre signatures de méthodes, lui-même basé sur un calcul de distances entre types dans le graphe d'héritage de *Java*. Ce mécanisme de recherche est expliqué en section 4.

2.2 Visualisation des résultats

Les résultats d'une requête sont généralement présentés sous la forme d'une liste. Cette représentation très simple, bien qu'elle permette d'ordonner les résultats, ne fournit que très peu d'information à l'utilisateur. Il n'est par exemple pas immédiat pour l'utilisateur de répondre à des questions telles que :

- Est-ce que les deux premiers résultats sont très proches de la requête et les suivants beaucoup plus éloignés ? S'il existe un saut dans les réponses, où se situe-t-il ?
- Quelles sont les réponses similaires entre elles ? Si tel résultat répond à mon problème, quels sont les autres résultats qui ont le plus de chance de me satisfaire également ?

Pour aider l'utilisateur à répondre à ces questions, le système présenté dans cet article accompagne la liste des résultats d'une représentation de ces derniers et de la requête dans un plan. Dans cette représentation, plus une méthode est proche de la requête, plus le point la représentant est proche de la position de la requête.

3 REPRÉSENTATION DES DONNÉES

Les données que nous traitons sont particulières étant donné qu'il s'agit de méthodes du langage *Java*. Notre approche a pour objectif de prendre en compte l'ensemble des bibliothèques du langage *Java*. Mais, pour plus de clarté, les tests et exemples proposés tout au long de cet article portent sur un sous-ensemble du langage *Java*. Il s'agit en fait d'une partie des paquetages *java.util* et *java.lang*. Le graphe d'héritage des classes incluses dans ce sous-ensemble est présenté sur les figures 1, 2 et 3. Il est à noter que ces trois graphes d'héritage sont reliés entre eux par les interfaces `Serializable` et `Cloneable`. Enfin, en *Java*, toutes les classes étendent la classe `Object`. Ceci n'est pas explicitement mentionné dans les figures. Cependant, on considère de façon implicite dans ces figures et dans le reste de l'article que toute classe, qui n'étend aucune classe de façon explicite, étend de façon implicite la classe `Object`.

Comme indiqué en introduction, la recherche s'effectue sur la signature des méthodes qui est composée de trois éléments : le type (ou la classe) auquel la méthode appartient, le type du résultat retourné par cette méthode, et le type des paramètres utilisés par la méthode. Il est à noter qu'une méthode s'applique à un objet (sauf dans le cas d'une méthode statique[7]). Le type de cet objet correspond au type auquel la méthode appartient. Nous considérons également cet objet comme un paramètre de la méthode. En effet, dans le cas d'une méthode qui n'est pas statique, l'objet sur lequel est appliqué la méthode influe sur le résultat retourné par cette dernière.

La première étape du processus consiste à proposer une représentation des signatures des méthodes. Ainsi, chaque méthode est caractérisée par un triplet $(C, R, \{P_1 \dots P_n\})$ qui est défini de la façon suivante :

- C représente la classe sur laquelle est invoquée la méthode,
- R représente le type de retour de la méthode,
- $\{P_1 \dots P_n\}$ est un ensemble composé des types des paramètres de la méthode.

Dans le cas où une méthode n'est pas statique, l'objet sur lequel est appliqué la méthode fait partie des paramètres. Donc, son type (C) fait également partie de l'ensemble composé des types des paramètres. Chaque méthode du langage de programmation peut s'exprimer sous la forme d'un tel vecteur (*cf.* exemple 1).

Exemple 1 : `Object getElementAt(int i)`

La méthode `getElementAt` de la classe `Vector` est une méthode non statique qui prend en paramètre un nombre entier (`int`) et retourne un objet (`Object`). Sa signature est alors représentée de la façon suivante :

`(Vector, Object, {Vector, int})`

La requête de l'utilisateur est décrite d'une façon similaire. Une requête est un triplet représentant une signature de méthode comme décrit précédemment. Cependant, le premier élément de ce triplet, c'est-à-dire la classe de la

[6] Le terme de distance n'est pas utilisé dans son sens strict d'un point de vue mathématique, étant donné que toutes les propriétés ne sont pas satisfaites (*e.g.* la symétrie).

[7] Identifiée dans le langage *Java* par le mot *static* dans la signature de la méthode.

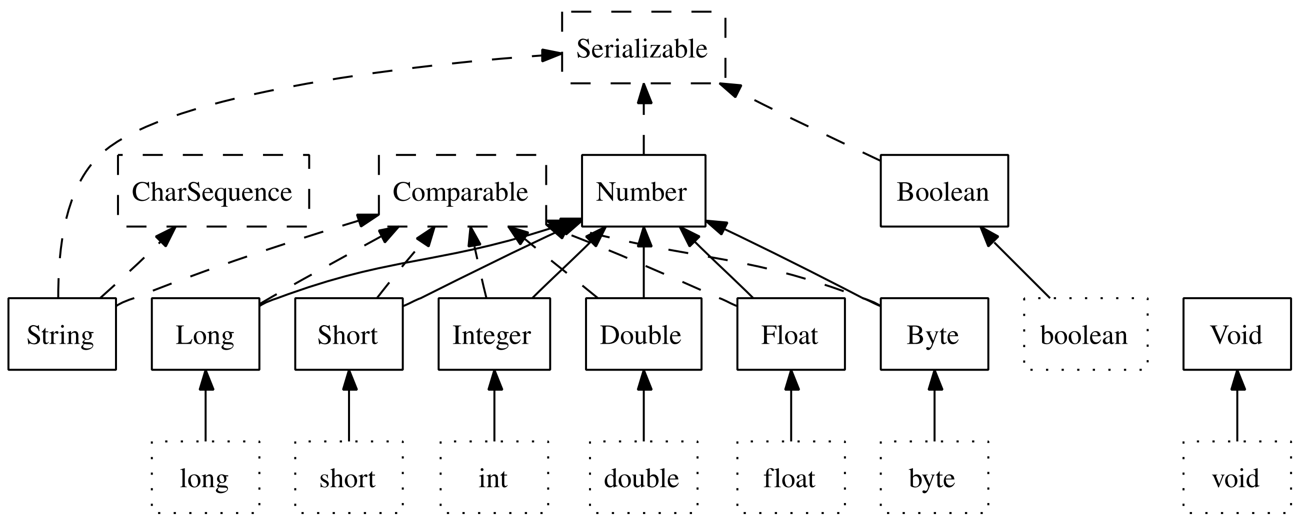

FIG. 1 – Graphe d'héritage de certains types primitifs

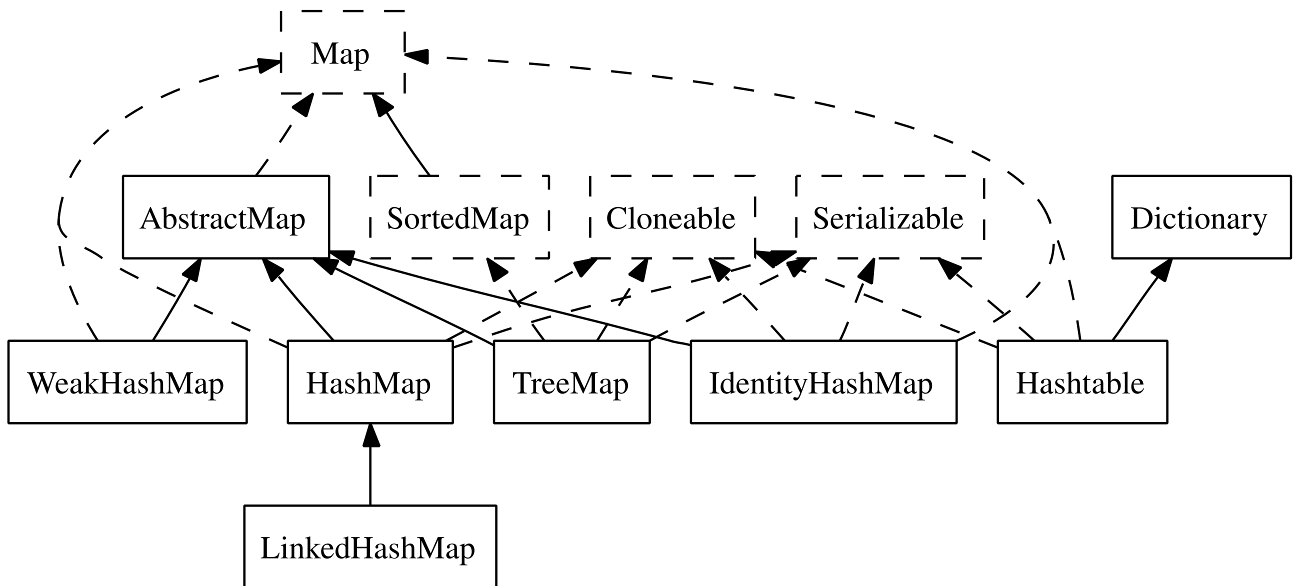

FIG. 2 – Graphe d'héritage de classes implémentant l'interface `Map`

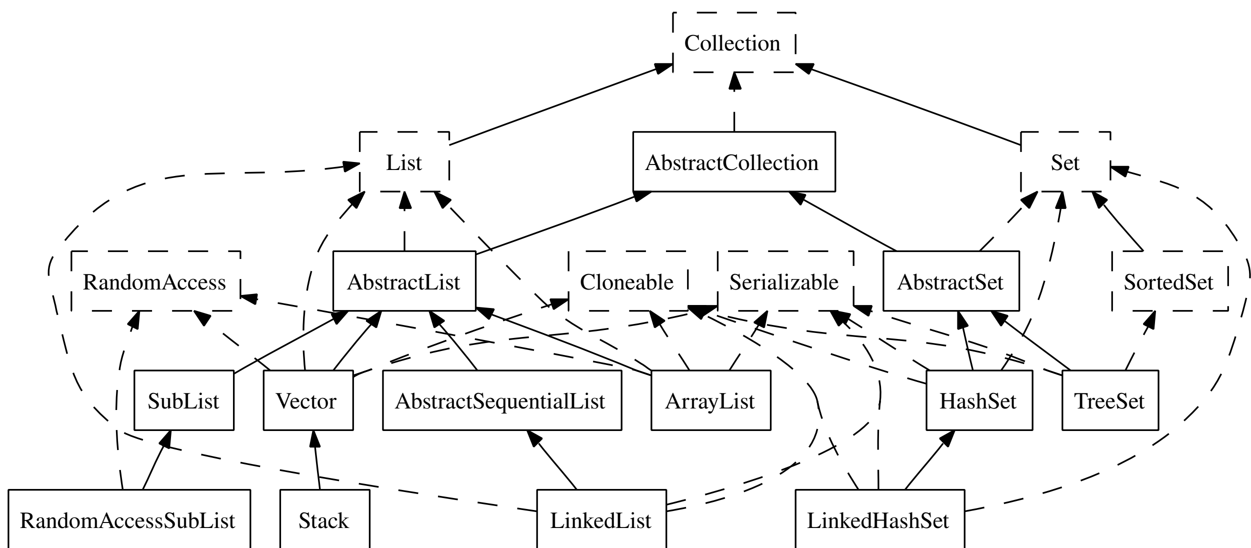

FIG. 3 – Graphe d'héritage de classes implémentant l'interface `Collection`

méthode, est optionnel. On utilise alors le point d'interrogation pour spécifier cette absence de contrainte sur le type de l'objet sur lequel la méthode doit s'appliquer (*cf.* exemple 2). Ce « joker » ne peut être utilisé que pour la classe de la méthode recherchée. En effet, les auteurs doutent de l'utilité de rechercher une méthode dont le résultat n'a pas d'importance. Par ailleurs, l'utilisation du point d'interrogation n'est pas utile dans la liste des paramètres car cette liste n'est pas exhaustive (elle contient les paramètres obligatoires, mais les méthodes retournées en réponse à la requête peuvent en contenir d'autres).

Exemple 2 : Représentation d'une requête

Une requête correspondant à des méthodes retournant un objet (type `Object`) et prenant en paramètres un vecteur (type `Vector`) et un nombre entier (type primitif `int`) peut s'exprimer de la façon suivante :

$$(?, \texttt{Object}, \{\texttt{Vector}, \texttt{int}\})$$

Si on souhaite obtenir des méthodes s'appliquant uniquement à des vecteurs, il convient alors de le spécifier explicitement. La requête devient alors :

$$(\texttt{Vector}, \texttt{Object}, \{\texttt{Vector}, \texttt{int}\})$$

Dans la suite de ce document, la notation $P_1, \ldots, P_n \rightarrow R$ représente la signature $(C, R, \{P_1 \ldots P_n\})$ pour laquelle la classe C n'est pas spécifiée.

4 MÉCANISME DE RECHERCHE

Comme indiqué en introduction, la plupart des outils de recherche de méthodes se basent sur une recherche par mots-clés. Les rares outils se basant sur la signature des méthodes recherchent soit uniquement les méthodes ayant cette exacte signature, soit les méthodes dont la signature est un sous-type de la requête (*cf.* rappel 1). Ainsi, dans le deuxième cas et avec le graphe d'héritage présenté en figure 1, si la recherche porte sur des méthodes prenant en paramètre un objet de type `Float` et retournant un objet de type `Integer`, le résultat de la recherche est composé de méthodes dont le paramètre est un objet de type `Float`, `Number`, `Comparable` ou `Serializable`, et la valeur de retour un objet de type `Integer` ou `int`.

Rappel 1 : sous-typage

$A \rightarrow B$ est un sous-type de $A' \rightarrow B'$
si et seulement si
A' est un sous-type de A et B est un sous-type de B'

Le principe de recherche que nous proposons dans cet article est différent. Il se base sur un calcul de distance entre les signatures des méthodes. Le résultat d'une recherche est alors une liste de méthodes retournées dans l'ordre croissant de leur distance à la requête jusqu'à atteindre un seuil fixé. Revenons à l'exemple de la recherche d'une méthode prenant en paramètre un objet de type `Float` et retournant un objet de type `Integer`. D'après le rappel 1, `float` → `int` n'est pas un sous-type de `Float` → `Integer`. Cependant, avec notre système, une méthode ayant pour signature `float` → `int` fait partie du résultat de la requête `Float` → `Integer` car sa signature est « proche » de celle de la requête.

4.1 Calcul des distances

La distance entre les méthodes est calculée à partir de leur signature. Cette distance est obtenue à partir de la distance entre les entités intervenant dans la signature. Dans le langage *Java*, une entité peut être une classe, une interface ou un type primitif[8]. Nous commençons donc par définir le calcul des distances entre deux entités.

4.1.1 Distance entre deux entités

La distance entre deux entités est obtenue à partir des chemins qui existent entre ces deux entités dans le graphe d'héritage. À chaque chemin est associé un coût. Ainsi, la distance entre deux entités est le coût minimum des chemins entre ces deux entités. La distance entre deux entités e_1 et e_n s'obtient donc de la façon suivante :

$$\text{cost}(e_1, e_n) = \min_k \text{cost}(\text{path}_k(e_1, e_n)) \quad (1)$$

où k est l'indice désignant les différents chemins possibles.

Chaque chemin est composé de sous-chemins de type unique et de longueur variable :

$$\begin{aligned} \text{path}_k(e_1, e_n) = \quad & \text{subpath}(e_1, e_l) \bullet \ldots \\ & \bullet \text{subpath}(e_i, e_j) \bullet \ldots \\ & \bullet \text{subpath}(e_{n-m}, e_n) \end{aligned} \quad (2)$$

$$\begin{aligned} \text{cost}(\text{path}_k(e_1, e_n)) = \quad & \text{cost}(\text{subpath}(e_1, e_l)) + \ldots \\ & + \text{cost}(\text{subpath}(e_i, e_j)) + \ldots \\ & + \text{cost}(\text{subpath}(e_{n-m}, e_n)) \end{aligned} \quad (3)$$

Le type d'un sous-chemin correspond à la relation d'héritage qui existe entre les différents nœuds le composant. Ainsi, de la classe `Number` au type primitif `int` il existe un chemin de type `extendedBy` car la classe `Number` est étendue par la classe `Integer`, elle-même étendue par le type primitif `int`. Il existe quatre types de chemins : `extends`, `extendedBy`, `implements` et `implementedBy`. La longueur d'un chemin correspond au nombre d'arcs qui le composent. Ainsi, la longueur du chemin direct de la classe `Number` au type primitif `int` est égale à 2. À partir de ces définitions, on détermine par exemple que le chemin le plus direct du type primitif `int` à l'interface `Serializable` (obtenu en passant par les classes `Integer` et `Number`) est composé d'un sous-chemin de type `extends` de longueur 2 et d'un sous-chemin de type `implements` de longueur 1.

[8] Tel que les entiers (`int`), les caractères (`char`) ou encore les booléens (`boolean`).

Le coût affecté à un sous-chemin de type unique entre les entités e_i et e_j est calculé en réalisant la somme du coût lié au type et du coût lié à la longueur du sous-chemin :

$$\begin{aligned} \text{cost}(\text{subpath}(e_i, e_j)) = \; & \text{type}_t\text{-cost} \\ & + (j - i) \times \text{arc}_t\text{-cost} \end{aligned} \tag{4}$$

où t est l'indice désignant le type du sous-chemin (ext pour un sous-chemin de type `extends`, $impl$ pour un sous-chemin de type `implements`, $extBy$ pour un sous-chemin de type `extendedBy` et $implBy$ pour un sous-chemin de type `implementedBy`). $(j - i)$ représente la longueur du sous-chemin, c'est-à-dire le nombre d'arcs parcourus pour aller de l'entité e_i à e_j. Suivant le type du sous-chemin, on dispose de différentes valeurs pour le coût lié au type et celui lié aux arcs. Ainsi, type_t-cost est le coût de base lié au type du chemin et arc_t-cost est le coût unitaire associé à **un** arc de ce type de chemin. Dans l'absolu, les coûts associés aux différents types de chemin ne sont pas égaux. Cependant, dans la première version du prototype présenté en section 6, ils le sont tous.

Exemple 3 : coût entre les entités `int` et `Serializable`

Le chemin de coût minimum entre les entités `int` et `Serializable` passe par les entités `Integer` et `Number`. Son coût est alors obtenu par le calcul suivant :

$$\begin{aligned} \text{cost} = \; & \text{type}_{ext}\text{-cost} + 2 \times \text{arc}_{ext}\text{-cost} \\ & + \text{type}_{impl}\text{-cost} + 1 \times \text{arc}_{impl}\text{-cost} \end{aligned} \tag{5}$$

La formule 4 a été choisie de façon à pouvoir rendre le coût d'un chemin de type unique mais de longueur élevée, inférieur au coût d'un chemin court mais composé de sous-chemins de types différents. Elle garantit aussi que les coûts des chemins de même type sont proportionnels à leur longueur.

4.1.2 *Distance entre signatures*

La distance entre deux ensembles d'entités E et F est la somme de la plus petite distance d'un élément e_i de E à un élément f_j de F et de la distance de l'ensemble $E \setminus \{e_i\}$ à l'ensemble $F \setminus \{f_j\}$. L'ensemble F doit être de taille supérieure ou égale à E. Dans le cas où l'ensemble F est plus grand que E, la différence entre la taille de F et celle de E est ajoutée au résultat final.

Exemple 4 : distance entre ensembles

Soit les ensembles $E = \{e_1, e_2\}$ et $F = \{f_1, f_2, f_3, f_4\}$ avec les distances suivantes :

	f_1	f_2	f_3	f_4
e_1	7	3	5	4
e_2	2	1	8	9

$$\text{distance}(E, F) = 1 + (4 + (|F| - |E|)) = 7$$

On rappelle que $|E|$ et $|F|$ représentent respectivement la taille des ensembles E et F.

Ainsi, la distance entre deux signatures (C_1, R_1, P_1) et (C_2, R_2, P_2) est définie comme étant la somme des distances de C_2 à C_1, de R_1 à R_2 et de P_2 à P_1. Par définition, la distance de ?[9] à n'importe quelle entité est nulle.

4.2 Appariement avec la requête

On applique le calcul de distances entre méthodes à notre cadre applicatif qu'est la recherche de méthodes dans un langage de programmation. On utilise alors les distances précédemment définies afin de calculer l'appariement, pour chaque méthode du langage, avec la requête de l'utilisateur. Chaque méthode obtient un score qui correspond à sa distance par rapport à la requête. On classe alors ces méthodes par scores croissants, la première étant la plus pertinente et la dernière la moins pertinente. Il est intéressant de noter que, pour une requête donnée, toutes les méthodes du langage ayant autant ou plus de paramètres que la requête, sont des réponses possibles. Seules les distances de ces méthodes à la requête changent. Il est donc nécessaire de définir un seuil afin de filtrer les méthodes et de ne conserver que les plus pertinentes, c'est-à-dire celles qui semblent apporter une réponse cohérente à la requête. Ainsi, la requête Q retourne des méthodes classées dans l'ordre croissant de leur distance à Q jusqu'à atteindre un seuil qui est actuellement fixé de façon empirique.

5 VISUALISATION DES RÉSULTATS

Une fois les résultats trouvés, il reste à les restituer efficacement à l'utilisateur au travers d'une interface graphique. Une première solution, basique et largement utilisée dans le domaine de la recherche d'information (RI), consiste à les afficher sous la forme d'une liste de résultats ordonnée selon leur distance par rapport à la requête (selon le calcul donné en section 4.1.2). Une telle interface est présentée sur la figure 4.

Dans un second temps, nous souhaitons proposer une visualisation cartographique des résultats, c'est-à-dire une visualisation au sein de laquelle le placement des résultats reflète leurs proximités les uns par rapport aux autres (*i.e.* un placement « sémantique »). Pour cela, il

[9] Utilisé lorsque la classe C de la requête n'est pas spécifiée.

1. Boolean Boolean.valueOf(boolean)
2. Boolean Boolean.Boolean(boolean)
3. boolean Boolean.equals(Object)
4. String Boolean.toString(boolean)
5. Boolean Boolean.valueOf(String)
6. Boolean Boolean.Boolean(String)
7. boolean Boolean.toBoolean(String)
8. boolean Boolean.getBoolean(String)
9. boolean String.regionMatches(boolean, int, String, int, int)
10. boolean AbstractCollection.remove(Object)

FIG. 4 – Visualisation des résultats sous la forme d'une liste

semble alors approprié d'utiliser une méthode permettant de projeter les données sur un espace 2D à partir de leur matrice de distances d. Chaque valeur d_{ij} de cette matrice est le minimum entre la distance du résultat M_i à M_j et celle du résultat M_j à M_i. Ces distances sont calculées de façon similaire à celles employées pour le calcul des distances entre les méthodes et la requête.

Il existe alors de nombreuses techniques envisageables mais leurs investigations ne font pas partie des objectifs initiaux de cet article. De plus, le cadre applicatif envisagé ici ne concerne qu'un très faible nombre de données, ce qui permet de ne pas se soucier de la complexité des différentes méthodes. Nous avons alors choisi d'utiliser la projection de Sammon qui est une technique largement employée dans de nombreux domaines depuis les années 70 [Sammon 1969]. Cette méthode est basée sur la minimisation de la fonction d'erreur suivante :

$$E = \frac{1}{\sum_{i<j} d_{ij}} \sum_{i<j}^{N} \frac{(d_{ij} - \delta_{ij})^2}{d_{ij}} \qquad (6)$$

où d_{ij} est la distance entre la méthode i et la méthode j calculée à la phase précédente. L'ensemble des termes d_{ij} constituent alors la matrice symétrique des distances entre les différentes méthodes. δ_{ij} est la distance euclidienne entre les positions des méthodes i et j dans l'espace 2D de représentation.

En termes de visualisation, cette technique présente cependant deux inconvénients : elle n'est pas déterministe si on suppose le cas classique d'une initialisation aléatoire des points sur l'espace 2D ; et il est possible que la minimisation de la fonction d'erreur E nous amène dans des minima locaux. Nous contournons alors ces deux problèmes avec la solution triviale suivante : nous fixons, au sein du prototype, 10 initialisations aléatoires et pour chaque requête, nous gardons uniquement le résultat offrant l'erreur minimale. Il est à noter que les distances dans l'espace 2D (δ_{ij}) ne respectent pas exactement les distances entre les méthodes (d_{ij}) pour la simple et bonne raison qu'il n'existe pas de projection exacte d'un espace à N dimensions (avec $N > 2$) sur un plan 2D. Cependant, l'utilisation du prototype développé permet de se rendre compte de l'utilité d'une représentation cartographique.

6 PROTOTYPE

Afin de valider notre démarche, un premier prototype, nommé *JMBrowser*, a été réalisé pour visualiser et évaluer les résultats fournis. Ce prototype a été testé sur les classes des figures 1, 2 et 3. Il faut cependant noter qu'il s'agit d'une première version qui est amenée à évoluer. Par ailleurs, ce prototype a pour but de valider l'intérêt de l'approche proposée, ce qui explique que l'optimisation du temps de calcul ne fasse pas partie des priorités.

6.1 Architecture et fonctionnement

Le prototype est décomposé en deux parties. La première, réalisée en *Prolog*, se charge de la représentation des données sous forme vectorielle ainsi que du calcul de l'ensemble des distances (entre les méthodes et la requête aussi bien qu'entre les méthodes elles-mêmes en vue de la partie visualisation). La seconde partie, développée en *Java*, est l'interface de recherche proposée à l'utilisateur. Elle comprend l'expression de la requête ainsi que la visualisation des résultats.

6.2 Résultats

La figure 5 présente les résultats fournis par le prototype *JMBrowser* pour la requête (`Boolean`, `Boolean`, {`boolean`}). L'utilisateur recherche alors des méthodes de la classe `Boolean`, prenant un paramètre de type `boolean` et rendant en résultat un objet de type `Boolean`. Une fois la requête effectuée, le prototype fournit à l'utilisateur une liste de résultats (partie droite de la figure) et leur représentation sous forme cartographique (partie gauche de la figure).

Liste de résultats. Les résultats sont affichés sous la forme d'une liste triée selon leur distance par rapport à la requête. Les différents champs de la liste de résultats sont : un numéro servant d'identifiant pour la carte 2D, la signature de la méthode (type de retour, nom de la classe, nom de la méthode et type des paramètres), ainsi que les distances à la requête. Cependant, ce dernier ne sert actuellement qu'à des fin de tests du prototype. Il est aussi prévu d'y afficher les commentaires *JavaDoc* associés aux différentes méthodes.

Carte des résultats. Les résultats sont aussi représentés sous forme cartographique selon l'algorithme décrit en section 5. Chaque point représente alors un résultat dont l'identifiant (un numéro) est affiché à côté du point. Un zoom est automatiquement appliqué sur la carte afin de maximiser l'utilisation de l'espace d'affichage. Sur cette carte, la requête n'est pas représentée par un point mais par un cercle centré sur la position de la requête. Par

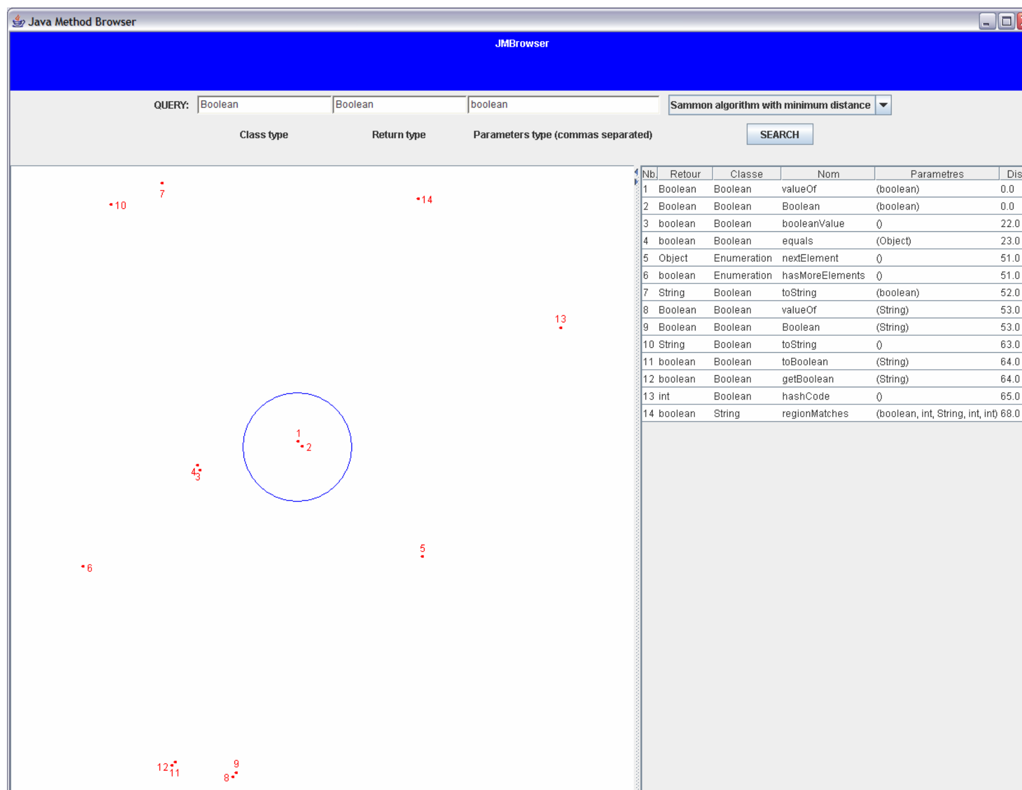

FIG. 5 – Prototype *JMBrowser* fournissant une visualisation cartographique des résultats

construction, la valeur du rayon de ce cercle correspond approximativement à la valeur de la distance associée à une relation de type `extends` dans le graphe d'héritage.

Interprétation des résultats. D'après la requête exprimée sur la figure 5, l'utilisateur cherche *a priori* à créer un objet de type `Boolean` ayant la même valeur que le paramètre utilisé (de type `boolean`). Le prototype rend deux résultats situés au centre du cercle et répondant donc exactement à la requête. Ces deux résultats sont le constructeur `Boolean` et la méthode statique `valueOf` (à laquelle l'utilisateur n'aurait peut-être pas pensé mais qui répond à ses besoins). On remarque aussi que les résultats ayant des signatures très similaires sont très proches sur la carte (*e.g.* les résultats 3 et 4 ou encore les résultats 11 et 12).

6.3 Évaluation

L'évaluation d'un tel système est toujours une tâche délicate mais nécessaire. Dans le domaine de la RI, les mesures traditionnellement utilisées sont basées sur les notions de rappel (R) et de précision (P) [Kent 1955]. Nous rappelons ci-dessous les notations ensemblistes de ces deux mesures. Pour une requête donnée, on note MP l'ensemble des méthodes pertinentes dans l'ensemble du langage et MR l'ensemble des méthodes retournées à l'utilisateur. Ainsi,

$$P = \frac{|MP \cap MR|}{|MR|} \qquad R = \frac{|MP \cap MR|}{|MP|}$$

Ces mesures peuvent être calculées pour différentes tailles de la liste des résultats, définies par la valeur du DCV[10]. Dans notre cas, ce seuil est indispensable étant donné que toutes les méthodes répondent à toutes les requêtes. En effet, seule leur distance par rapport à la requête change. Cependant, pour pouvoir utiliser cette approche, il faut disposer de l'ensemble MP. Cela nécessite, pour une requête donnée, de constituer manuellement cet ensemble. De plus, il faut alors définir des critères objectifs permettant de dire si une méthode répond de façon pertinente à la requête. Ne disposant pas actuellement de ces annotations de pertinence, l'évaluation de notre prototype est reportée à de prochains travaux.

7 TRAVAUX SIMILAIRES

7.1 Outils de recherche de méthodes *Java*

À la connaissance des auteurs, en ce qui concerne spécifiquement les outils de recherche de méthodes *Java*, il n'existe que deux outils vraiment opérationnels : *JavaDoc*[11] et le mécanisme de recherche fourni par l'IDE[12] *Eclipse*[13].

JavaDoc, à partir d'un ensemble de fichiers source *Java*, génère un document avec hyperliens contenant la majorité des méthodes apparaissant dans les fichiers

[10] Document cut off values
[11] `http://fr.wikipedia.org/wiki/Javadoc`
[12] *Integrated Development Environment* — environnement de développement
[13] `http://www.eclipse.org/`

source. Ce document peut être assimilé à un dictionnaire dont les entrées sont les classes des méthodes. Si la génération d'un « dictionnaire » est simple, la recherche de méthodes est, quant à elle, plus compliquée. En effet, pour retrouver une méthode, l'utilisateur doit connaître la classe à laquelle elle appartient et parcourir la page associée à cette classe.

L'IDE *Eclipse* fournit un outil de recherche dans les fichiers source *Java*. Cet outil permet entre autres de rechercher une méthode en associant des mots-clés à différents champs tels que : la classe d'une méthode, son nom ou ses paramètres. Les mots-clés pris en compte correspondent à la majorité des mots apparaissant dans la définition de cette méthode dans le code source. Le problème de cette approche, lors d'une recherche par signature de méthode, est qu'il faut donner le type exact de la méthode (au nombre de paramètres près).

7.2 Travaux de recherche similaires

Il existe de nombreux travaux de recherche sur la réutilisation de composants logiciels [Mili 1998, Frakes 2005]. Parmi ceux-ci certains concernent la recherche de méthodes (ou de fonctions dans le cas de langages fonctionnels) à partir de leur signature.

Mickaël Rittri s'intéresse à la recherche de fonctions dans un langage proche de ML[14] (Lazy ML) [Rittri 1991]. Le mécanisme qu'il propose se base sur une relation d'équivalence entre le type des fonctions. Deux types sont considérés comme équivalents s'ils ne diffèrent que par l'ordre des paramètres ou par le niveau de *curryfication* [Schönfinkel 1924]. Lors d'une recherche, son système retourne l'ensemble des fonctions dont le type appartient à la même classe d'équivalence que la requête. Ceci implique que l'utilisateur doit connaître de façon exacte le nombre et le type des paramètres de la fonction qu'il recherche.

Toujours dans le domaine de la recherche de fonctions, Colin Runciman et Ian Toyn proposent une approche basée sur le polymorphisme de type [Runciman 1991]. Cela permet de définir une relation de sous-typage basée sur un ordre partiel. Le type `T` est un sous-type du type `U` si et seulement s'il est possible d'unifier `U` à `T` par instanciation des variables de type apparaissant dans `U`. Les résultats d'une recherche avec leur système est l'ensemble des fonctions dont le type est un sous-type de la requête.

S'appuyant sur un système de fichiers logique (LISFS [Padioleau 2003]), Benjamin Sigonneau et Olivier Ridoux ont développé un outil de recherche de méthodes *Java* basé, entre autres, sur leur signature [Sigonneau 2006]. Du point de vue de ces dernières, le mécanisme proposé se base sur l'ordre partiel entre les signatures dû au sous-typage exprimé par l'héritage. Lorsqu'une requête concerne uniquement la signature des méthodes, les signatures des résultats sont des sous-types de la signature de la requête. En outre, leur mécanisme permet de prendre en compte des mots-clés lors de la recherche. De plus, l'utilisation de LISFS permet une navigation progressive dans les résultats par raffinements successifs de la requête.

Cependant, tous ces travaux rendent en résultat d'une requête uniquement des réponses « exactes » (à une relation de sous-typage près). À la différence de ces travaux, le mécanisme proposé dans cet article retourne un ensemble de méthodes « similaires » à la requête. Il fournit également à l'utilisateur une représentation graphique lui permettant d'évaluer rapidement la pertinence des différents résultats. Cette approche permet d'effectuer des recherches même dans le cas où l'utilisateur n'a pas une connaissance exacte du type ou d'un type plus général de la méthode recherchée.

8 CONCLUSION

Cet article propose une démarche pragmatique pour la recherche de méthodes dans un langage de programmation, et plus particulièrement dans le langage *Java*. Des solutions sont alors apportées à différents problèmes allant de la représentation des données à leur visualisation. La méthode proposée est basée sur un calcul de distances entre les signatures des méthodes. Ainsi, il est possible de retrouver des réponses exactes (méthodes ayant une signature identique à celle de la requête), mais aussi des réponses approchées. Dans ce dernier cas, il s'agit de méthodes rapidement exploitables par le programmeur. Par ailleurs, nous proposons une visualisation cartographique des résultats. Cette représentation est basée sur une représentation planaire des distances entre les différents résultats. Elle permet ainsi d'identifier rapidement les résultats intéressants par rapport à la requête effectuée.

De nombreuses améliorations peuvent être apportées à l'approche proposée dans cet article. Ainsi, concernant le mécanisme de recherche, il faudrait remettre en question le calcul de certaines distances entre classes (notamment lorsque le chemin dans le graphe d'héritage passe par la classe `Object`). Il pourrait également être utile de pondérer les paramètres selon leur importance dans la requête, et d'analyser le contexte de la recherche afin d'améliorer notre approche. Enfin, il serait intéressant de prendre en compte les classes génériques (*Generics* ou *templates*) de la version 1.5 de *Java*. Du point de vue de la visualisation, des tests plus approfondis doivent être mis en œuvre afin d'adapter efficacement l'algorithme de placement des résultats. Bien entendu, l'ergonomie du prototype (qui ne fait pas partie des objectifs initiaux en termes de visualisation) peut être amplement améliorée. Pour des travaux futurs, une piste intéressante serait de coupler notre approche avec une recherche par mots-clés afin d'améliorer les résultats obtenus.

[14] `http://fr.wikipedia.org/wiki/ML_(langage)`

REMERCIEMENTS

Nous remercions Fabienne Moreau et Stéphanie Neveu pour leurs relectures de nos travaux ; Yoann Dubreuil pour les discussions techniques que nous avons partagées avec lui ; Pierrette Josse et Laurence Chouisnard pour leur soutien tout au long de nos travaux ; ainsi que Cédric Piednoir pour l'originalité de ses remarques.

RÉFÉRENCES

[Di Cosmo 1993] Di Cosmo R., Deciding Type Isomorphisms in a type assignment framework. *Journal of Functional Programming*, 3(3) :485–525. Special Issue on ML.

[Frakes 2005] Frakes W. B. et Kang K., Software Reuse Research : Status and Future. *IEEE Transactions on Software Engineering*, 31(7) :529–536.

[Kent 1955] Kent A., Berry M. M., Luehrs, JR. F. U. et Perry J. W., Operational Criteria for Designing Information Retrieval Systems. *American Documentation*, 6 :93–101.

[Mili 1998] Mili A., Mili R. et Mittermeir R. T., A Survey of Software Reuse Libraries. *Annals of Software Engineering*, 5 :349–414.

[Padioleau 2003] Padioleau Y. et Ridoux O., A Logic File System. *Proceedings of the USENIX Annual Technical Conference*.

[Rittri 1991] Rittri M., Using Types as Search Keys in Function Libraries. *Journal of Functional Programming*, 1(1) :71–89.

[Runciman 1991] Runciman C. et Toyn I., Retrieving Reusable Software Components by Polymorphic Type. *Journal of Functional Programming*, 1(2) :191–211.

[Sammon 1969] Sammon, Jr J. W., A Non-Linear Mapping for Data Structure Analysis. *IEEE Transactions on Computers*, C-18(5) :401–409.

[Schönfinkel 1924] Schönfinkel M., Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92 :305–316. Traduit en anglais par Stefan Bauer-Mengelberg et republié sous le titre « On the Building Blocks of Mathematical Logic » dans [van Heijenoort 1967, pages 355–366].

[Sigonneau 2006] Sigonneau B. et Ridoux O., Indexation multiple et automatisée de composants logiciels. *Technique et Science Informatiques*, 25(1) :9–42.

[van Heijenoort 1967] van Heijenoort J., editor *From Frege to Gödel*. Harvard University Press.