Refactoring Browser, SmallLint et RewriteTool

Dr. Ducasse

ducasse@iam.unibe.ch
http://www.iam.unibe.ch/~ducasse/

Avec ce numéro spécial sur la ré-ingénierie des applications, nous allons pouvoir vous montrer quelques-uns des excellents outils disponibles en Smalltalk qui facilitent la ré-ingénièrie des applications à objets[DDN02]. Ce mois-ci nous présentons le Refactoring Browser et ses outils. Cet article complète l'article que nous avions écrit il y a quelques mois sur SUnit. L'importance de ces outils est telle qu'ils ne sont pas spécifiques à un environment Smalltalk comme Squeak ou VisualWorks mais existent dans pratiquement *tous* les Smalltalks. Le Refactoring Browser comme SUnit dont les premières versions ont été développés en Smalltalk ont une grande influence sur l'évolution des environnements de programmation comme récemment Eclipse

Notons que le Refactoring Browser et SUnit sont en synergie car l'un permet de transformer du code et l'autre de spécifier des tests qui peuvent identifier les problèmes introduits lors de transformation manuelle qui sont aussi nécessaires.

Le Refactoring Browser est un des précurseurs dans le domaine de la transformation de code avec préservation du comportement. Les auteurs du Refactoring Browser ont participé au livre Improving the Design of Existing Code de Marting Fowler[FBBOR]. Depuis 1995, il est disponible gratuitement pour VisualWorks à www.refactory.com/RefactoringBrowser/. Le Refactoring Browser est disponible pour la plupart des environnements Smalltalk comme VisualAge Smalltalk (www.software.ibm.com/ad/smalltalk), l'excellent Dolphin Smalltalk (www.object-arts.com/) et Squeak (www.squeak.org). Son importance est telle que la toute dernière version de VisualWorks (www.cincom.smalltalk.com), la version 7, a remplacé tous ces browseurs de code par une version améliorée et complètement intégrée du Refactoring Browser. C'est cette version que nous utilisons dans cet article et que vous pouvez obtenir dans la version non commerciale de VisualWorks (www.cincomsmalltalk.com:8080/CincomSmalltalkWiki/) [VW].

1. Les Refactorings

Un refactoring est une transformation de code conversant son comportement [FBBOR]. Un refactoring spécifie des pré-conditions qui définissent les conditions qui vont permettre au refactoring de transformer de manière *sécurisée* un morceau de code.

Illustration. Prenons par exemple le refactoring "Abstraction de Variable d'Instance", il crée des accesseurs pour une variable d'instance et remplace tous les accès directs à cette variable d'instance par un appel à l' accesseur adéquat. Une des pré-conditions est donc de tester qu'une méthode de même nom que l'accesseur n'existe pas dans la classe ou ses superclasses.

Le refactoring "Abstraction de Variable d'Instance" permet de transformer *très rapidement* et de manière *sûre* un attribut en un attribut paresseux, c'est-à-dire, dont la valeur ne sera définie que si nécessaire lors de la première lecture. Mais montrons les étapes ! Imaginons la classe

`Widget` qui définit la variable d'instance `background` accédée directement par la méthode `redraw` sur cette même classe.

```
Object subclass: #Widget
  instanceVariableName: 'background'

Widget>>redraw:anImage
  ...
  self invalidateRegion: background size.
  ...
```

La première étape pour rendre la variable d'instance paresseuse est d'abstraire la variable, c'est-à-dire de créer les accesseurs si nécessaire et sans masquer de possibles méthodes. La seconde étape est de changer *tous* les accès directs aux variables d'instances par des appels aux accesseurs.

```
Widget>>background
  ^ background

Widget>>background: anImage
  background: anImage

Widget>>redraw:anImage
  ...
  self invalidateRegion: self background size.
  ...
```

La troisième étape est de changer la définition de l'accesseur en lecture comme suit :

```
Widget>>background
    background isNil
        ifTrue: [background := Image default].
    ^ background
```

Finalement il faut s'assurer que l'initialisation de la variable d'instance `background` est bien `nil`. En Smalltalk, `nil` est la valeur par défaut des variables d'instances donc il suffit de s'assurer que cette valeur n'est pas changée lors de l'initialisation des instances par une méthode de type `initialize`. Notez que les deux dernières phases ne peuvent pas être automatisées simplement. Cependant, obtenir les accès directs ou invocations aux accesseurs en écriture est une aide.

Notez qu'abstraire une variable sans l'aide d'un outil spécialisé est fastidieux et prompt à introduire des erreurs : Il faut vérifier si les accesseurs existent, éventuellement les créer en vérifiant que l'on ne masque pas une méthode ayant une sémantique différente, ensuite il faut identifier *tous* les accès à cette variable et les remplacer par les appels adéquats. Manquer un seul accès peut introduire des bugs.

Le livre de Martin Fowler [FBBOR] auquel les auteurs du Refactoring Browser ont participé décrit un certain nombre de refactorings de manière *informelle* et certains des refactorings *ne sont pas automatisables* par un outil car nécessitant une analyse que seul un humain peut conduire. Le Refactoring Browser pour sa part implémente une quinzaine de refactorings dont la sémantique permet leur automatisation, il est complètement intégré à la façon de développer en Smalltalk et de naviguer le code.

2. Le Refactoring Browser

La page du Refactoring Browser originelle est http://st-www.cs.uiuc.edu/users/brant/Refactory/. On y trouve en particulier les articles [RBJ1] et [RBJ2] décrivant les différents refactorings implantés et une description des outils l'accompagnant.Une page plus actuelle est http://www.refactory.com/RefactoringBrowser/. Mais le Refactoring Browser étant devenu le browser officiel de VisualWorks dans sa version 7.0. Il suffit de prendre la version non commerciale de VisualWorks pour l'avoir.

Exemple. Une méthode est l'unité de réutilisation et doit donc représenter une seule opération sémantique et être composée d'actions d'un même niveau d'abstraction. Fowler et al dans [FBBOR] présentent une liste de symptômes que le code peut présenter et qui indique clairement la nécessité de le refactoriser. Parmi ces symptômes on trouve de longues méthodes découper par des commentaires.

Ici nous avons pris la méthode `openCascade:type:name:` de la classe `UIBuilder` bienqu'elle ne soit pas problèmatique. La figure suivante montre la sélection d'une partie de la méthode en vue de son extraction comme une nouvelle méthode nommée `openNormalWindow:`.

```
UIBuilder>>openCascade: aWindowSpec type: sizeType name: specName
...
   ifTrue:
     [self window
       openIn: (self window cascadedPositionRectangle: aWindowSpec
bounds extent)
       withType: #normal]
   ifFalse:
     [aWindowSpec sizeAutoSave ifTrue: [bindings at: #__autoSave put:
#size].
     ...
```

Une fois la partie de texte sélectionnée et l'extraction demandée, le Refactoring Browser analyse si cette extraction est possible, si l'extraction est possible il vérifie que ce morceau de code *n'existe pas déjà* sous forme d'une méthode dans la classe, auquel cas il propose de remplacer la sélection par un appel à cette méthode avec les bonnes valeurs. Lorsque la méthode n'existe pas, il calcule le nombre de paramètres nécessaires à la méthode et demande un nom de méthode. Notez que l'on peut changer l'ordre des paramètres si on le souhaite à l'aide des fléches.

```
UIBuilder>>openCascade: aWindowSpec type: sizeType name: specName
...
   ifTrue: [self openNormalWindow: aWindowSpec]
   ifFalse:
     [aWindowSpec sizeAutoSave ifTrue: [bindings at: #__autoSave put:
#size]...


UIBuilder>>openNormalWindow: aWindowSpec
  ^self window
   openIn: (self window cascadedPositionRectangle: aWindowSpec
bounds extent)
```

```
  withType: #normal
```

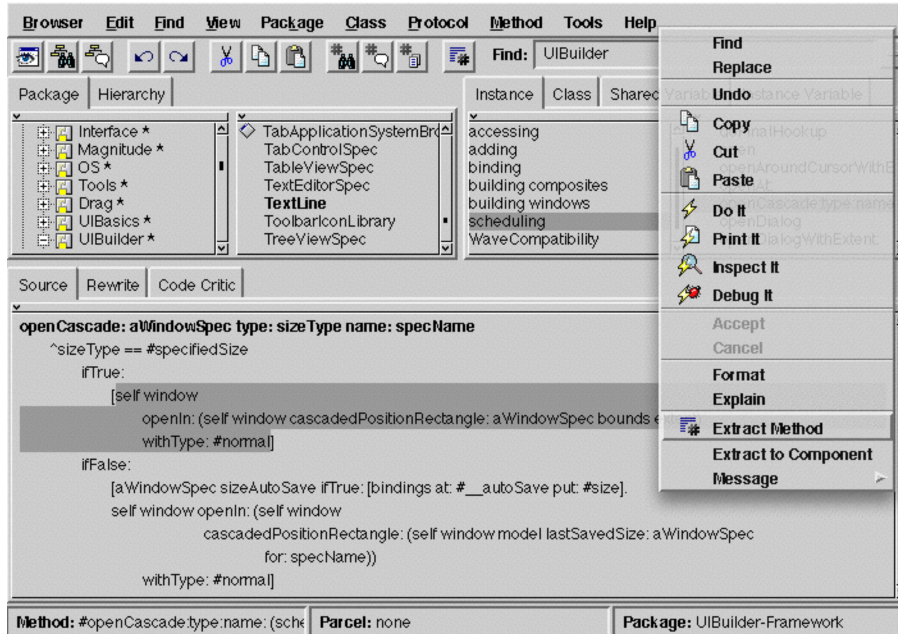
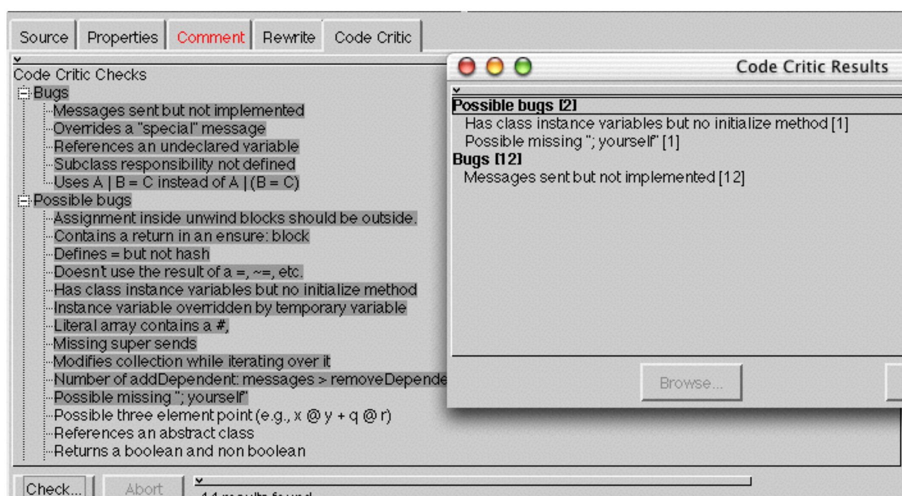


Figure 1 Extraction d'un morceau de code en méthode.

Refactorings. Le Refactoring Browser définit les refactorings nommés dans la table 1. Nous n'entrons pas dans les détails relatifs à la distinction entre variables d'instances et variables de classes. La description complète est disponible dans son site web [RB] et expliquée dans [RBJ1] et [RBJ2]. Certains de ces refactorings peuvent sembler triviaux comme ajouter un paramètre à une méthode mais n'oubliez pas que le code doit continuer de fonctionner après l'application du refactoring. Dans le cas d'Ajoute un Paramètre, il faut donc spécifier une valeur par défaut qui sera utilisée comme valeur du nouveau paramètre lors de l'invocation de la méthode et que l'on pourra modifier ultérieurement. Notez de plus que le Refactoring Browser offre l'intéressante possibilité d'enlever (undo) les refactorings appliqués.

Table 1: Quelques Refactorings Implantés dans le Refactoring Browser

Insère classe dans hiérarchie	Descend/Monte méthode dans hiérarchie
Enlève classe	Enlève méthode
Ajoute/Enlève/Renomme variable d'instance	Ajoute/Enlève paramètre
Abstrait/Concrètise variable d'instance	Converti temporaire en variable d'instance
Crée Accesseurs	Déplace dans autre classe
Renomme méthode	Extrait code comme méthode
Descend/Monte variable d'instance	Inclut messages



3. SmallLint et le Rewrite Tool

Le Refactoring Browser est accompagné de deux autres outils très intéressants : SmallLint qui est un outil analysant du code Smalltalk et détectant des bugs ou de possibles erreurs, et RewriteTool qui permet d'exprimer la réécriture de code par le biais de reconnaissance d'expressions (pattern matching) sur des arbres de syntaxes abstraites.

SmallLint. SmallLint est un outil d'analyse de code. Il cherche à identifier une soixantaine de problèmes possibles allant du simple bug, au bug possible, en passant par du code inutile ou l'identification de méthodes trop longues. Pour obtenir, il suffit de cliquer sur l'onglet code critic du Refactoring Browser. Vous obtenez une fenêtre comme celle présentée dans la Figure suivante qui montre le résultat de l'application de quelques règles sur quelques classes. Pour vous en servir, vous devez sélectionner une classe (non montrée sur dans la figure), choisir quelles jeux de règles vous voulez appliquer, sélectionner ensuite les règle (montrées en grisé) et finalement presser Check... Une fois les résultats vous pouvez avoir accès aux méthodes suspectes en cliquant sur les lignes qui composent le résultat.

Certaines sociétés qui développent en Smalltalk obligent les développeurs à invoquer SmallLint systématiquement avant de versionner leur code. Notons que les règles peuvent être particularisées et de nouvelles règles peuvent être ajoutées au jeu de règles existant. La définition des règles utilise la reconnaissance de code (pattern matching) proposé par le RewriteTool que nous allons voir maintenant.

RewriteTool. RewriteTool est un outil de réécriture de code basé sur la définition de reconnaissance de formes (pattern matching) appliquée sur arbres de syntaxe abstraites. Une documentation plus complète est disponible à http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser/Rewrite.html. Cet outil de réécriture de code est particulièrement utile lorsque du code doit être transformé de manière répétitive.Prenons un exemple réel, entre les versions 2.5 et 3.0 de VisualWorks une nouvelle méthode nommée `at:ifAbsentPut:` fut introduite sur la classe Dictionary. Elle rendait obsolete le code suivant aDictionary at: key ifAb-

sent: `[aDictionary at: key put: value]` qui mettait une valeur dans un dictionnaire lorsque cette valeur n'était pas déjà présente. Avec cette nouvelle méthode l'ancienne expression peut être écrite de la manière suivante: `aDictionary at: key ifAbsentPut: value`. Cependant transformer de telles expressions est laborieux car le block [...] peut contenir d'autres expressions. Le RewriteTool permet d'exprimer un schéma d'identification et un schéma de transformation comme montré dans la figure 3. Cet outil va analyser le code sélectionné puis montrer à l'utilisateur les transformations qui peuvent être rejetées ou appliquées.

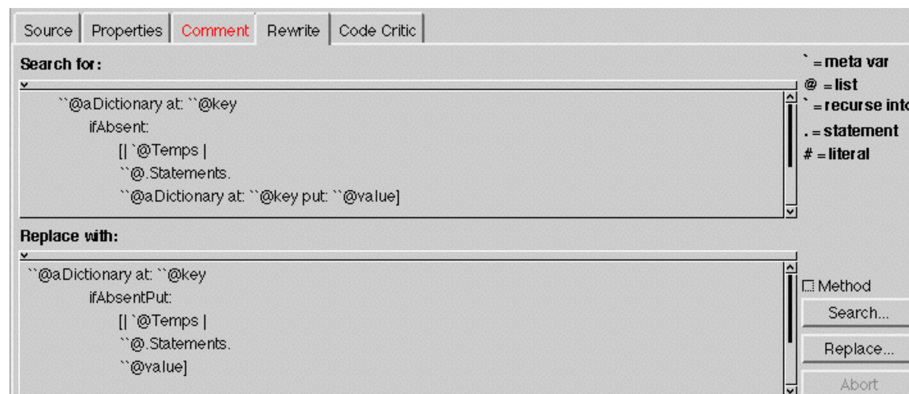


Figure 3 RewriteTool: pour rééecrire du code satisfaisant certaines formes

Pour obtenir le RewriteTool depuis le Refactoring Browser, il suffit de cliquer sur l'onglet Rewrite.

Variable. Un schéma peut contenir des variables en utilisant le backquote ou accent grave. Ainsi `` `key `` représente n'importe quelle variable mais pas une expression.

Liste. Pour représenter une expression potentiellement complexe, on utilise @ qui représente une liste. Ainsi `` ` @key `` peut représenter aussi bien une variable simple comme `temp` qu'une expression comme `(aDict at: self keyForDict)`. Par exemple, `` | `@Temps | `` reconnait une liste de variable temporaires. Le point . reconnait une instruction dans une séquence de code. `` `@.Statements `` reconnaît une liste d'instructions. `` foo `@message: `@args `` reconnaît n'importe quel message envoyé à foo.

Récursion. Maintenant si l'on veut que la reconnaissance s'effectue aussi à l'intérieur de l'expression il faut doubler la quote. La second quote représente la récursion du schéma cherché. Ainsi `` ` `@object foo `` reconnaît foo envoyé à n'importe quel objet mais aussi pour chaque reconnaissance regarde si une reconnaissance existe dans la partie représentée par la variable `` ` `@object ``.

Littéraux. # représente les littéraux par exemple, `` `#literal `` reconnaît n'importe quel littéral comme 1, #(), "unechaine" ou #unSymbol.

Des exemples. Si l'on veut identifier les expressions `aDict at: aKey ifAbsent: aBlock` dans lesquelles les variables peuvent être des expressions composées, nous écrivons l'expression suivante ``` ``@aDict at: ``@aKey ifAbsent: ``@aBlock ```. Une telle expression identifie par exemple les expressions suivantes:

```
instVarMap at: aClass name ifAbsent: [oldClass instVarNames]
deepCopier references at: argumentTarget ifAbsent: [argumentTarget]
bestGuesses at: anInstVarName ifAbsent: [self typesFor: anInstVarName]
object at: (keyArray at: selectionIndex) ifAbsent: [nil]
```

4. Conclusion

Bien que peu connu, les outils et les concepts développés autour du langage Smalltalk ont une réelle influence sur notre vie: que ce soit l'utilisation de la souris, l'invention du multi fenêtrage et d'environnements de programmation, la programmation à objets ou la programmation agile [Beck99], SUnit et les refactorings [FBBOR]. Nous espérons vous avoir donner envie de regarder de plus près ces outils qui permettent un développement sécurisé et efficace ainsi qu'une aide vitale lors de la ré-ingéniérie des applications objets [DDN02].

Références et liens

[Beck99] Kent Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.

[DDN02] S. Demeyer, S. Ducasse et O. Nierstrasz, Object-Oriented Reengineering Patterns, Morgan Kaufman Publishers, 2002.

[FBBOR] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[RB] http://www.refactory.com/RefactoringBrowser/, http://st-www.cs.uiuc.edu/users/brant/Refactory/

[RBJ1] D. Roberts, J. Brant and R. Johnson, "Why every Smalltalker should use the Refactoring Browser, Smalltalk Report, SIGS Press, http://st-www.cs.uiuc.edu/users/droberts/homePage.html#refactoring

[RBJ2]D. Roberts, J. Brant and R. Johnson, "A Refactoring Tool for Smalltalk", TAPOS, vol. 3, no. 4, 1997, pp. 253-263, http://st-www.cs.uiuc.edu/~droberts/tapos/TAPOS.htm

[SUnit] http://www.xprogramming.com/software.htm

[VW] www.cincom.com/smalltalk, http://www.cincomsmalltalk.com:8080/CincomSmalltalkWiki/, http://www.cincomsmalltalk.com:8080/CincomSmalltalkWiki/VW+7+White+Paper+2