

Chapter 1

Validation et Transformations de code automatique en Pharo

SmallLint est un outil qui analyse du code Squeak et qui détecte des bogues 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.

1.1 Analyse qualitative de code avec SmallLint

SmallLint est un outil d'analyse de code. Il permet d'identifier une soixantaine de problèmes possibles allant du simple bogue, à la prévision de bogue, en passant par la détection de code inutile ou l'identification de méthodes trop longues. SmallLint met en évidence des problèmes au niveau de méthodes ou de classes qui utilisent l'héritage, et détecte certaines erreurs.

Pour ouvrir cet outil, exécutez l'expression `LintDialog open` ; vous obtenez une fenêtre comme celle qui est présentée figure qui montre le résultat de l'application de quelques règles sur les classes.

`@@ here rules @@` Pour vous en servir, vous devez choisir les jeux de règles que vous souhaitez appliquer (dans le panneau, en haut à gauche), sélectionner les règles (panneau, en bas à gauche), les catégories (panneau du milieu), les classes (panneau de droite), et finalement presser `Run`. Une fois que tout est affiché, vous pouvez avoir accès aux méth-

odes suspectes en cliquant sur les lignes qui détaillent le résultat. Certaines sociétés imposent aux développeurs d'invoquer systématiquement SmallLint avant de délivrer leur code. Notons que les règles peuvent en être particularisées et qu'il est possible d'en ajouter de nouvelles au jeu existant. La définition des règles utilise la reconnaissance de code (pattern matching) proposé par le RewriteTool que nous allons maintenant étudier.

1.2 Identification de code avec RewriteTool

RewriteTool est un outil de réécriture de code basé sur la définition de reconnaissance de formes (pattern matching), appliquée sur des arbres de syntaxes abstraites. Une documentation plus complète est disponible à <http://st-www.cs.uiuc.edu/brant/RefactoringBrowser/Rewrite.html>.

Il semble que Squeak ne dispose pas actuellement d'interface graphique pour la réécriture du code, mais uniquement pour identifier des morceaux de code.

Cet outil de réécriture de code est particulièrement utile lorsqu'on doit transformer d'une manière répétitive du code. On peut représenter dans les schémas (patterns) de reconnaissance des variables, des listes, des instructions récursives et des littéraux.

- **Variable.** Un schéma peut contenir des variables en utilisant le back-quote ou accent grave. Ainsi, ``key` représente n'importe quelle variable, mais pas une expression.
- **Liste.** Pour représenter une expression potentiellement complexe, on utilise `@` qui caractérise une liste. Ainsi, `@key` peut représenter aussi bien une variable simple comme `temp` qu'une expression comme `(aDict at: self keyForDict)`. Par exemple, `|@Temps|` reconnaît une liste de variables temporaires. Le point `.` reconnaît une instruction dans une séquence de code. `@.Statements` reconnaît une liste d'instructions. Par exemple, `foo @message: @args` reconnaît n'importe quel message envoyé à `foo`.
- **Récursion.** Pour que la reconnaissance s'effectue aussi à l'intérieur de l'expression, il faut doubler la quote. La seconde quote représente la récursion du schéma cherché. Ainsi, ```@object foo` reconnaît `foo`, à quelque objet qu'il soit envoyé, mais observe également pour chaque reconnaissance si une reconnaissance est possible dans la partie représentée par la variable ```@object`.
- **Littéraux.** `\#` représente les littéraux ; ainsi, `\#literal` reconnaît n'importe quel littéral, par exemple `1`, `\#()`, `"unechaine"` ou `\#unSymbol`.

1.3 Des exemples d'identification de schémas

Si l'on veut identifier les expressions de type aDict at: aKey ifAbsent: aBlock dans lesquelles les variables peuvent être des expressions composées, on écrit 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]
```

Comme l'interface en Squeak ne permet pas encore de sélectionner les classes sur lesquelles on veut travailler, le système analyse les 1 934 classes et quelque 42 869 méthodes qui sont disponibles dans la distribution de base, ce qui peut sensiblement ralentir le traitement.

Voici quelques exemples d'expressions qui pourraient être avantageusement transformées :

```
| `@Temps | ``@.Statements. ``@Boolean ifTrue: [↑false]. ↑true
| `@Temps | ``@.Statements. ↑``@Boolean not
``@object not ifTrue: ``@block
``@object ifFalse: ``@block.
```

```
rule := RBUnderscoreAssignmentRule new.
environment := BrowserEnvironment new forPackageNames: #('PackageA'
'PackageB' ...).
SmalllintChecker runRule: rule onEnvironment: environment.
rule open
```

```
ORLintBrowser
  openRule: (RBCompositeLintRule rules: (RBCompositeLintRule
rulesGroupedFor: RBSpellingRule) name: 'Spelling')
  environment: (BrowserEnvironment new forPackageNames: #('Kernel'
'Collections-Abstract'))
```