Chapter 1

Smalllint: static analysis in Pharo

1.1 Ensuring Quality

Good design practices are fundamental requisites to address software inherent properties (e.g., complexity, conformity, changeability). But smells are often introduced unintentionally by developers during early software development or software maintenance. For example, a software designer may adopt well-known established practices during initial design; however, such design may indicate certain structural deficiencies or smells that have arised during the process. Also, software developers who are tasked with software maintenance (e.g., develop new features or fix bugs) may introduce smells into the code. It is important in both cases to address the smells as to reduce the technical debt and maintain a high structural quality of the software. Awareness of smells enable designers to make well-informed design decisions and developers to avoid introducing smells in the software.

As defined by Martin Fowler, smells are certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring. Basically, three types of smells can be found in source code at different levels: architectural, design and implementation. The architectural level includes smells such as "god package" and "cyclical dependency between packages". The design (or micro-architectural) level includes smells such as "cyclic hierarchy" and "large abstraction". Finally, the implementation level includes smells such as "improper name length" and "variables having constant value". Smalllint aims the detection of smells at design and

implementation level, so this chapter is limited to such types of smells.

1.2 SmallLint in a Nutshell

False positives

1.3 Existing SmallLint Rules

Bugs

message send but not implemented

Potential Bugs

read before set

Design Flaws

subclass

Coding Idiom Violation

todo instead collection

Optimization

Style

empty catger empty class comment

Varia

Table

1.4 Defining your own rules

@@autofixing rules

Block rules

Abstract Syntax Tree-Based Rules

Defining a simple rules

1.5 Conclusion

1.6 Junk

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 récriture de code par le biais de reconnaissance d'expressions (pattern matching) sur des arbres de syntaxes abstraites.

1.7 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, Ãă 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 fenÃłtre 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 Ãă gauche), sélectionner les règles (panneau, en bas Ãă gauche), les catégories (panneau du milieu), les classes (panneau de droite), et finalement presser Âń Run Âż. Une fois que tout est affiché, vous pouvez avoir accès aux méthodes 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 Ãłtre 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.

1.8 Identification de code avec RewriteTool

RewriteTool est un outil de récriture 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 Ãă http://st-www.cs.uiuc.edu/ brant/RefactoringBrowser/ Rewrite.html.

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

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

- 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 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 | reconnaÃőt une liste de variables temporaires. Le point . reconnaÃőt une instruction dans une séquence de code.`@.Statements reconnaÃőt une liste d'instructions. Par exemple, foo '@message: `@args reconnaÃőt n'importe quel message envoyé Ãă foo.

- Récursion. Pour que la reconnaissance s'effectue aussi Ãă l'intérieur de l'expression, il faut doubler la quote. La seconde quote représente la récursion du schéma cherché. Ainsi, ``@object foo reconnaÃőt foo, Ãă 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 reconnaÃőt n'importe quel littéral, par exemple 1, \#(), "unechaine" ou \#unSymbol.

1.9 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 Ãłtre 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 Ãłtre 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'))
```