

Chapter 1

Smallint: 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 arisen 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". Smallint aims the detection of smells at design and

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

1.2 Existing SmallLint Rules

Bugs

Method has no timeStamp (RBMethodHasNoTimeStampRule): This smell arises when a method has no timestamp. For proper versioning, every method should have a timestamp.

Messages sent but not implemented (RBSentNotImplementedRule): This smell arises when a message is sent by a method, but no class in the system implements such a message. This method sent will certainly cause a `doesNotUnderstand: message` when they are executed. Further this rule checks if messages sent to self or super exist in the hierarchy, since these can be statically typed.

Subclass responsibility not defined (RBSubclassResponsibilityNotDefinedRule): This rule checks if all `subclassResponsibility` methods are defined in all leaf classes. If such a method is not overridden, a `subclassResponsibility` message can occur when this method is called.

Sends super new initialize (RBSuperSendsNewRule): This rule checks for a method that wrongly initializes an object twice. Contrary to other Smalltalk implementations, Pharo automatically calls `#initialize` on object creation. For example, a warning is raised when the statement `self new initialize` is found in a method.

Style

Class variable capitalization (RBClassVariableCapitalizationRule): This smell arises when class or pool variable names do not start with an uppercase letter, which is a standard style in Smalltalk.

Temporary variable capitalization (RBTemporaryVariableCapitalizationRule): This smell arises when a temporary or argument variable does not start with a lowercase letter, which is a standard style in Smalltalk.

Redundant class name in selector (RBClassNameInSelectorRule): This smell arises when the class name is found in a selector. This is redundant since to call the you must already refer to the class name. For example, `#openHierarchyBrowserFrom:` is a redundant name for `HierarchyBrowser`.

Instance variable capitalization (RBInstanceVariableCapitalizationRule):

This smell arises when instance variable names (in instance and class side) do not start with an lowercase letter, which is a standart style in Smalltalk.

Potential Bugs

Returns a boolean and non boolean (RBReturnsBooleanAndOtherRule): This smell arises when a method return a boolean value (true or false) and return some other value such as (nil or self). If the method is suppose to return a boolean, then this signifies that there is one path through the method that might return a non-boolean. If the method doesn't need to return a boolean, it should be probably rewritten to return some non-boolean value since other programmers reading the method might assume that it returns a boolean.

Defines = but not hash (RBDefinesEqualNotHashRule): This smell arises when a class defines `#=` also and not `#hash`. If `#hash` is not defined then the instances of the class might not be able to be used in sets since equal element must have the same hash.

Coding Idiom Violation

Literal array contains a #, (RBLiteralArrayContainsCommaRule): This smell arises when literal arrays contain the `#`, symbol. It can be done accidentally by new developers when trying to separate elements. Elements in arrays are separated by space, ex: `#(1 2 3)`. Elements in dynamic arrays are separated by `#`, symbol, ex: `1. 2. 3`

No class comment (RBNoClassCommentRule): This smell arises when a class has no comment. Classes should have comments to explain their purpose, collaborations with other classes, and optionally provide examples of use.

Sends "questionable" message (RBBadMessageRule): This smell arises when methods send messages that perform low level things. You might want to limit the number of such messages in your application. Messages such as `#isKindOf:` can signify a lack of polymorphism. You can see which methods are "questionable" by editing the `RBBadMessageRule>>badSelectors` method. Some examples are: `#respondsTo:` `#isMemberOf:` `#performMethod:` and `#performMethod:arguments:`

Design Flaws

Methods equivalently defined in superclass (RBEquivalentSuperclassMethodsRule): This smell arises when a method is equivalent to its superclass

method. The methods are equivalent when they have the same abstract syntax tree, except for variables names. Such method does not add anything to the computation and can be removed since the superclass method have the same behaviour. Furthermore, the methods `#new` and `#initialize` are ignored once they are often overridden for compatibility with other platforms. The ignored methods can be edited in `RBEquivalentSuperclassMethodsRule`.

Excessive inheritance depth (`RBExcessiveInheritanceRule`): This smell arises when a deep inheritance is found (depth of ten or more), which is usually a sign of a design flaw. It should be broken down and reduced to something manageable. The defined inheritance depth can be edited in `RBExcessiveInheritanceRule>>inheritanceDepth`.

Inconsistent method classification (`RBInconsistentMethodClassificationRule`): This smell arises when a method protocol is not equivalent to the one defined in the superclass of such method class. All methods should be put into a protocol (method category) that is equivalent to the one of the superclass, which is a standard style in Smalltalk. Furthermore, methods which are extension in the superclass are ignored, since they may have different protocol name.

Long methods (`RBLongMethodsRule`): This smell arises when a long method is found (with 10 or more statements). Note that it counts statements, not lines. The defined number of statements can be edited in `RBLongMethodsRule`.

Excessive number of arguments (`RBExcessiveArgumentsRule`): This smell arises when a method contains a long number of argument (five or more), which can indicate that a new object should be created to wrap the numerous parameters. The defined number of arguments can be edited in `RBExcessiveArgumentsRule>>argumentsCount`.

Methods implemented but not sent (`RBImplementedNotSentRule`): This smell arises when a method is never sent. If a method is not sent, it can be removed. Furthermore, methods with pragmas and test methods are likely to be sent through reflection, thus such methods are ignored.

Excessive number of variables (`RBExcessiveVariablesRule`): This smell arises when a class has too many instance variables (10 or more). Such classes could be redesigned to have fewer fields, possibly through some nested object grouping. The defined number of instance variables can be edited in `RBExcessiveVariablesRule>>variablesCount`.

Refers to class name instead of "self class" (`RBRefersToClassRule`): This smell arises when a class has its class name directly in the source instead of "self class". The self class variant allows you to create subclasses without needing to redefine that method.

Excessive number of methods (`RBExcessiveMethodsRule`): This smell

arises when a large class is found (with 40 or more methods). Large classes are indications that it has too much responsibility. Try to break it down, and reduce the size to something manageable. The defined number of methods can be edited in `RBExcessiveMethodsRule>>methodsCount`.

Class not referenced (`RBClassNotReferencedRule`): This smell arises when a class is not referenced either directly or indirectly by a symbol. If a class is not referenced, it can be removed.

Instance variables defined in all subclasses (`RBInstVarInSubclassesRule`): This smell arises when instance variables are defined in all subclasses. Many times you might want to pull the instance variable up into the class so that all the subclasses do not have to define it.

Method defined in all subclasses, but not in superclass (`RBMissingSubclassResponsibilityRule`): This smell arises when a class defines a method in all subclasses, but not in itself as an abstract method. Such methods should most likely be defined as subclassResponsibility methods. Furthermore, this check helps to find similar code that might be occurring in all the subclasses that should be pulled up into the superclass.

Optimization

Instance variables not read AND written (`RBOnlyReadOrWrittenVariableRule`): This smell arises when an instance variable is not both read and written. If an instance variable is only read, the reads can be replaced by `nil`, since it could not have been assigned a value. If the variable is only written, then it does not need to store the result since it is never used. This check does not work for the data model classes since they use the `#instVarAt:put:` messages to set instance variables.

Method just sends super message (`RBJustSendsSuperRule`): This smell arises when a method just forward the message to its superclass. These methods can be removed.

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

1.3 Defining your own rules

@@autofixing rules

Block rules

Abstract Syntax Tree-Based Rules

Defining a simple rules

1.4 Conclusion

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

1.6 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 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 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.

1.7 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 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 | 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 Ã foo.
- Récursion. Pour que la reconnaissance s'effectue aussi Ã l'intérieur de l'expression, il faut doubler la quote. La seconde quote représente la récursion du schéma cherché. Ainsi, ``@object foo reconnaît foo, Ã 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.

1.8 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'))
```