Chapter 1

SmallLint: static analysis in Pharo

Being able to check that the code of your application follows certain rules is important to control its quality. Pharo offers SmallLint a tool originally developed by John Brant and Don Roberts to identify several families of problems that code may exhibit. SmallLint defines a list of static analyses grouped by topics and that you can run automatically on your code. In addition, in Pharo, package meta-data supports the application of SmallLint by letting the programmer flagging irrelevant or false positives. In this chapter we will @@

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 arose 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

SmallLint is a tool that analyses Pharo code and identifies bugs, design problems and other mismatches to recommanded idioms.

Style

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

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.

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.

Temporary variable capitalization (RBTemporaryVariableCapitalizationRule): This smell arises when a temporary or argument variable do not start with a 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 rewriten 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.

Design Flaws

Methods equivalently defined in superclass (RBEquivalentSuperclass-MethodsRule): 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 compatilbity with other platforms. The ignored methods can be edited in RBEquivalentSuperclassMethodsRule>>ignoredSelectors

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 standart style in Smalltalk. Furthermore, methods which are extension in the superclass are ignored, since they may have different protocol name.

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.

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.

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 edit in RBExcessiveMethodsRule>>methodsCount.

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>>longMethodSize.

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.

Instance variables defined in all subclasses (RBInstVarInSubclasses-Rule): 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 (RBMissingSub-classResponsibilityRule): 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.

Coding Idiom Violation

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: #respond-sTo: #isMemberOf: #performMethod: and #performMethod:arguments:

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

Overrides a "special" message (RBOverridesSpecialMessageRule): Checks that a class does not override a message that is essential to the base system. For example, if you override the #class method from object, you are likely to crash your image. In the class the messages we should not override are: ==, , class, basicAt:, basicAt:put:, basicSize, identityHash. In the class side the messages we should not override are: basicNew, basicNew, class, comment, name.

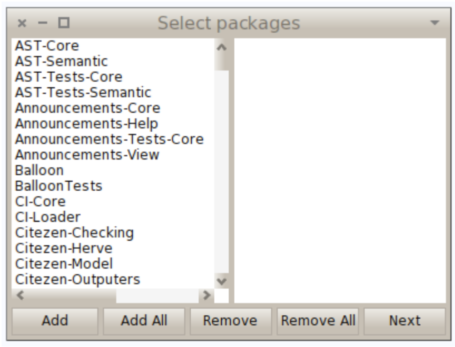
Messages sent but not implemented (RBSentNotImplementedRule): This smell arises when a message is sent by a method, but no class in the system implements such a message. This method sent will certainly cause a doesNotUnderstand: message when they are executed. Further this rule checks if messages sent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

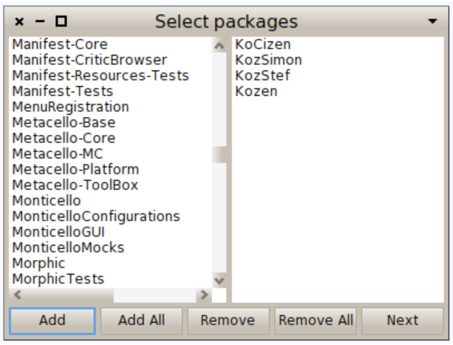
Subclass responsibility not defined (RBSubclassResponsibilityNotDefinedRule): This rule checks if all subclassResponsibility methods are defined in all leaf classes. if such a method is not overridden, a subclassResponsibility message can be occur when this method is called

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

1.3 Using Code Critics

You can invoke the Code Critic Browser via the Tools menu. We will execute it on one of our project Kozen whose purpose is to generate static web page based on scientific publications. We started from a project on which we never run the rules.





Selecting Rules. Once you have selected the packages on which you want to run the rules, you can select them as shown in Figure 1.1. Rules are sorted in different categories as explained in Section 1.2. By default running all the rules is a good idea.

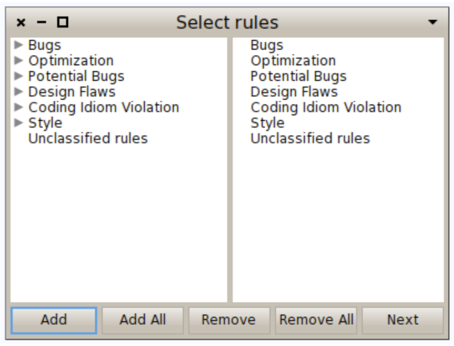


Figure 1.1: Selecting rules. By default running all the rules is a good idea.

First look at Results. Once the rules are run you get a browser showing you the results as shown in Figure 1.2. Once the rules are run, we obtain a set of rule violations, we have several possibilities:

- *Addressing the problem.* In such a case it can be wise to rerun the rule to verify that it has been addressed.

- *Marking the problem as a ToDo.* The point here is that tagging a violation as to-do makes sure that the violation will not show up as red warning when rules are checked again later. To-dos indicate that the package developer knows that there is an issue that it should be fixed later. Having Todos is a nice feature because it lets the developer decide when to address a problem while avoiding the tools to always report it as a problems.

- *Marking the violation as a tool error.* Indeed a rule may be wrong or irrelevant. We call such error a false positive. Marking a violation as a false positive makes sure that the next time the rule execution will be executed

With the Critics Browser we payed attention that once a developer runs and evaluates the violations (*i.e.*, addressing, marking them as todos or false positives), he can be sure that a new execution of the rules on the same code will not report again the same problems.

The Critics Browser shows the results are grouped by rule kinds. The top level label labelled selected Rules (FP: 0, ToDo: 0, Total: 326 means the following: we got a total of 326 rule violations. Since we just started to check a new project, we did not mark any violations as false positive[1], this is why we have FP: 0, and since we did not flag violations as point to address in the future we have ToDo: 0.

Figure 1.2 shows that in the rules related to style issues, the Kozen packages got two badly classified methods. Since moving the methods to a correct method category is easy, we did it and run the rules. We obtain then the situation described by Figure 1.3.

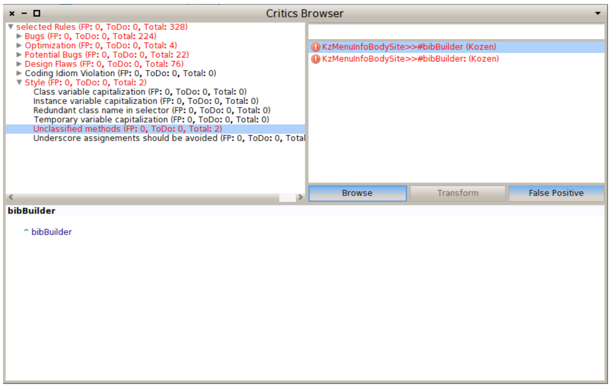
The Critics Browser shows the rules on the left pane. When one rule is selected, the violations appear in the right pane. You can search by typing in the top right input field. The pane at the bottom shows either the rule description or the entity exhibiting the violation.

A more interesting case is the 'Bugs' category as shown by Figure **??**

Banning a complete rule.

Banning a single critic.

[1] A false positive is said to a violation that was detected by a tool but that was not true.



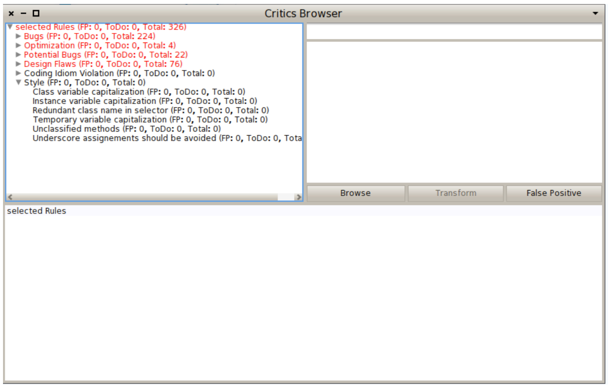
```
▼ selected Rules (FP: 0, ToDo: 0, Total: 328)
  ▶ Bugs (FP: 0, ToDo: 0, Total: 224)
  ▶ Optimization (FP: 0, ToDo: 0, Total: 4)
  ▶ Potential Bugs (FP: 0, ToDo: 0, Total: 22)
  ▶ Design Flaws (FP: 0, ToDo: 0, Total: 76)
  ▶ Coding Idiom Violation (FP: 0, ToDo: 0, Total: 0)
  ▼ Style (FP: 0, ToDo: 0, Total: 2)
      Class variable capitalization (FP: 0, ToDo: 0, Total: 0)
      Instance variable capitalization (FP: 0, ToDo: 0, Total: 0)
      Redundant class name in selector (FP: 0, ToDo: 0, Total: 0)
      Temporary variable capitalization (FP: 0, ToDo: 0, Total: 0)
      Unclassified methods (FP: 0, ToDo: 0, Total: 2)
      Underscore assignements should be avoided (FP: 0, ToDo: 0, Total

KzMenuInfoBodySite>>#bibBuilder (Kozen)
KzMenuInfoBodySite>>#bibBuilder: (Kozen)

bibBuilder

    ^ bibBuilder
```

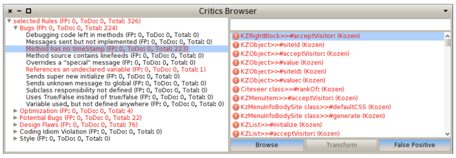
Figure 1.2: Browsing rule results. Two methods are not well classified in the class KzMenuInfoBoySite.



```
▼ selected Rules (FP: 0, ToDo: 0, Total: 326)
  ▶ Bugs (FP: 0, ToDo: 0, Total: 224)
  ▶ Optimization (FP: 0, ToDo: 0, Total: 4)
  ▶ Potential Bugs (FP: 0, ToDo: 0, Total: 22)
  ▶ Design Flaws (FP: 0, ToDo: 0, Total: 76)
  ▶ Coding Idiom Violation (FP: 0, ToDo: 0, Total: 0)
  ▼ Style (FP: 0, ToDo: 0, Total: 0)
      Class variable capitalization (FP: 0, ToDo: 0, Total: 0)
      Instance variable capitalization (FP: 0, ToDo: 0, Total: 0)
      Redundant class name in selector (FP: 0, ToDo: 0, Total: 0)
      Temporary variable capitalization (FP: 0, ToDo: 0, Total: 0)
      Unclassified methods (FP: 0, ToDo: 0, Total: 0)
      Underscore assignements should be avoided (FP: 0, ToDo: 0, Tota

selected Rules
```

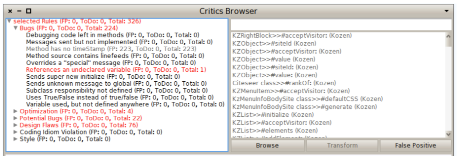
Figure 1.3: Addressing an issue on the spot and running the rules once again



```
▼ selected Rules (FP: 0, ToDo: 0, Total: 326)
  ▼ Bugs (FP: 0, ToDo: 0, Total: 224)
      Debugging code left in methods (FP: 0, ToDo: 0, Total: 0)
      Messages sent but not implemented (FP: 0, ToDo: 0, Total: 0)
      Method has no timeStamp (FP: 0, ToDo: 0, Total: 223)
      Method source contains linefeeds (FP: 0, ToDo: 0, Total: 0)
      Overrides a "special" message (FP: 0, ToDo: 0, Total: 0)
      References an undeclared variable (FP: 0, ToDo: 0, Total: 1)
      Sends super new initialize (FP: 0, ToDo: 0, Total: 0)
      Sends unknown message to global (FP: 0, ToDo: 0, Total: 0)
      Subclass responsibility not defined (FP: 0, ToDo: 0, Total: 0)
      Uses True/False instead of true/false (FP: 0, ToDo: 0, Total: 0)
      Variable used, but not defined anywhere (FP: 0, ToDo: 0, Total: 0)
  ▶ Optimization (FP: 0, ToDo: 0, Total: 4)
  ▶ Potential Bugs (FP: 0, ToDo: 0, Total: 22)
  ▶ Design Flaws (FP: 0, ToDo: 0, Total: 76)
  ▶ Coding Idiom Violation (FP: 0, ToDo: 0, Total: 0)
  ▶ Style (FP: 0, ToDo: 0, Total: 0)

KZRightBlock>>#acceptVisitor: (Kozen)
KZObject>>#siteId (Kozen)
KZObject>>#acceptVisitor: (Kozen)
KZObject>>#value (Kozen)
KZObject>>#siteId: (Kozen)
KZObject>>#value: (Kozen)
Citeseer class>>#rankOf: (Kozen)
KZMenuItem>>#acceptVisitor: (Kozen)
KzMenuInfoBodySite class>>#defaultCSS (Kozen)
KzMenuInfoBodySite class>>#generate (Kozen)
KZList>>#initialize (Kozen)
KZList>>#acceptVisitor: (Kozen)
```

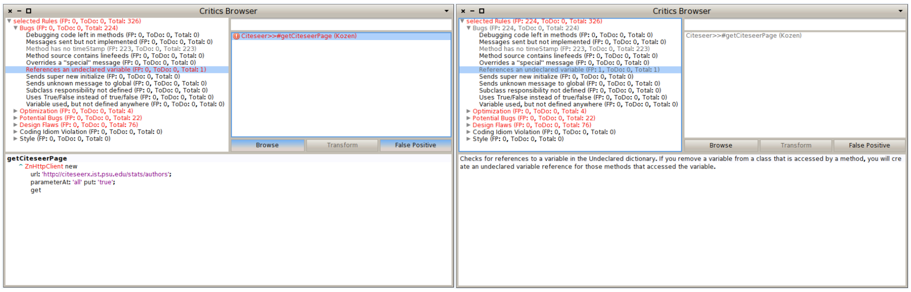
Figure 1.4: Looking at the 'Bugs' rule category.



```
▼ selected Rules (FP: 0, ToDo: 0, Total: 326)
  ▼ Bugs (FP: 0, ToDo: 0, Total: 224)
      Debugging code left in methods (FP: 0, ToDo: 0, Total: 0)
      Messages sent but not implemented (FP: 0, ToDo: 0, Total: 0)
      Method has no timeStamp (FP: 223, ToDo: 0, Total: 223)
      Method source contains linefeeds (FP: 0, ToDo: 0, Total: 0)
      Overrides a "special" message (FP: 0, ToDo: 0, Total: 0)
      References an undeclared variable (FP: 0, ToDo: 0, Total: 1)
      Sends super new initialize (FP: 0, ToDo: 0, Total: 0)
      Sends unknown message to global (FP: 0, ToDo: 0, Total: 0)
      Subclass responsibility not defined (FP: 0, ToDo: 0, Total: 0)
      Uses True/False instead of true/false (FP: 0, ToDo: 0, Total: 0)
      Variable used, but not defined anywhere (FP: 0, ToDo: 0, Total: 0)
  ▶ Optimization (FP: 0, ToDo: 0, Total: 4)
  ▶ Potential Bugs (FP: 0, ToDo: 0, Total: 22)
  ▶ Design Flaws (FP: 0, ToDo: 0, Total: 76)
  ▶ Coding Idiom Violation (FP: 0, ToDo: 0, Total: 0)
  ▶ Style (FP: 0, ToDo: 0, Total: 0)
```

```
KZRightBlock>>#acceptVisitor: (Kozen)
KZObject>>#siteId (Kozen)
KZObject>>#acceptVisitor: (Kozen)
KZObject>>#value (Kozen)
KZObject>>#siteId: (Kozen)
KZObject>>#value: (Kozen)
Citeseer class>>#rankOf: (Kozen)
KZMenuItem>>#acceptVisitor: (Kozen)
KzMenuInfoBodySite class>>#defaultCSS (Kozen)
KzMenuInfoBodySite class>>#generate (Kozen)
KZList>>#initialize (Kozen)
KZList>>#acceptVisitor: (Kozen)
KZList>>#elements (Kozen)
```

Figure 1.5: Banning the rule for the complete selected packages.



```
▼ selected Rules (FP: 0, ToDo: 0, Total: 326)
  ▼ Bugs (FP: 0, ToDo: 0, Total: 224)
      Debugging code left in methods (FP: 0, ToDo: 0, Total: 0)
      Messages sent but not implemented (FP: 0, ToDo: 0, Total: 0)
      Method has no timeStamp (FP: 223, ToDo: 0, Total: 223)
      Method source contains linefeeds (FP: 0, ToDo: 0, Total: 0)
      Overrides a "special" message (FP: 0, ToDo: 0, Total: 0)
      References an undeclared variable (FP: 0, ToDo: 0, Total: 1)
      Sends super new initialize (FP: 0, ToDo: 0, Total: 0)
      Sends unknown message to global (FP: 0, ToDo: 0, Total: 0)
      Subclass responsibility not defined (FP: 0, ToDo: 0, Total: 0)
      Uses True/False instead of true/false (FP: 0, ToDo: 0, Total: 0)
      Variable used, but not defined anywhere (FP: 0, ToDo: 0, Total: 0)
  ▶ Optimization (FP: 0, ToDo: 0, Total: 4)
  ▶ Potential Bugs (FP: 0, ToDo: 0, Total: 22)
  ▶ Design Flaws (FP: 0, ToDo: 0, Total: 76)
  ▶ Coding Idiom Violation (FP: 0, ToDo: 0, Total: 0)
  ▶ Style (FP: 0, ToDo: 0, Total: 0)
```

```
Citeseer>>#getCiteseerPage (Kozen)
```

```
getCiteseerPage
    ^ ZnHttpClient new
        url: 'http://citeseerx.ist.psu.edu/stats/authors';
        parameterAt: 'all' put: 'true';
        get
```

```
▼ selected Rules (FP: 224, ToDo: 0, Total: 326)
  ▼ Bugs (FP: 224, ToDo: 0, Total: 224)
      Debugging code left in methods (FP: 0, ToDo: 0, Total: 0)
      Messages sent but not implemented (FP: 0, ToDo: 0, Total: 0)
      Method has no timeStamp (FP: 223, ToDo: 0, Total: 223)
      Method source contains linefeeds (FP: 0, ToDo: 0, Total: 0)
      Overrides a "special" message (FP: 0, ToDo: 0, Total: 0)
      References an undeclared variable (FP: 1, ToDo: 0, Total: 1)
      Sends super new initialize (FP: 0, ToDo: 0, Total: 0)
      Sends unknown message to global (FP: 0, ToDo: 0, Total: 0)
      Subclass responsibility not defined (FP: 0, ToDo: 0, Total: 0)
      Uses True/False instead of true/false (FP: 0, ToDo: 0, Total: 0)
      Variable used, but not defined anywhere (FP: 0, ToDo: 0, Total: 0)
  ▶ Optimization (FP: 0, ToDo: 0, Total: 4)
  ▶ Potential Bugs (FP: 0, ToDo: 0, Total: 22)
  ▶ Design Flaws (FP: 0, ToDo: 0, Total: 76)
  ▶ Coding Idiom Violation (FP: 0, ToDo: 0, Total: 0)
  ▶ Style (FP: 0, ToDo: 0, Total: 0)
```

```
Citeseer>>#getCiteseerPage (Kozen)
```

```
Checks for references to a variable in the Undeclared dictionary. If you remove a variable from a class that is accessed by a method, you will cre
ate an undeclared variable reference for those methods that accessed the variable.
```

Figure 1.6: Looking at the undeclared variable bug (left) and Banning one single critic (right)

1.4 Defining Your Own Rules

Block Rules

Block rules use the Smalltalk reflective API. They can be created to find methods that should be not invoked, style consistence such capitalization or variable name length, class or method size, classes not commented, variables not referenced, instance variables defined in all subclasses, among others. In resume, every thing that is possible to do using the Smalltalk reflective API can be used in a block rule. This includes access to the Smalltalk model which allows the easy navigation through classes (and their superclasses and subclasses), methods, variables, arguments, comments, invocations, etc.

These rules are created by extending the class RBBlockLintRule.

TODO

- Two levels: class and method.

- Class level rules must implement the method #checkClass: and method level rules must implemente the method #checkMethod:

- Example

Abstract Syntax Tree-Based Rules

These rules are based on the Smalltalk abstract syntax tree (AST). They can be created to find assignments with no effect, weak use of the API (pieces of code can be more efficient or legible), among others. In summaty, these rules performs operation in AST nodes to find smells.

These rules are created by extending the class RBParseTreeLintRule.

- Class level rules must implement the method #checkClass: and method level rules must implemente the method #checkMethod:

- Example

Defining Simple Rules

1.5 Conclusion

1.6 Junk

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'))
```