A Critique on Code Critics

Angela Lozano *, Gabriela Arévalo **, and Kim Mens

Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium
alozano@soft.vub.ac.be

Universidad Abierta Interamericana
Av. Montes de Oca 745
Buenos Aires, Argentina
gabriela.b.arevalo@gmail.com

Université catholique de Louvain
Place Sainte Barbe 2
Louvain-la-Neuve, Belgium
kim.mens@uclouvain.be

Abstract. *Code critics* are a recommendation facility of the Pharo Smalltalk IDE. They signal controversial implementation choices such as code smells at class and method level. They aim to promote the use of good and standard coding idioms for increased performance or a better use of object-oriented constructs. This paper studies relations among code critics by analyzing co-occurrences of code critics detected on the Moose system, a large and mature Smalltalk application. Based upon this analysis, we present a critique on code critics, as a first step towards an improved grouping of code critics that identifies issues at a higher level of abstraction, by combining lower-level critics that tend to co-occur, as well as improvements in the definition of the individual critics.

Keywords: code critics, bad smells, co-occurrence, Smalltalk, Pharo, empirical software engineering

1 Introduction

A plethora of code recommendation tools exists to support developers when coding a software system. Whereas some of these recommendations remain at a high level of abstraction (*e.g.*, low coupling and high cohesion), others are much more specific (*e.g.*, 'classes should not have more than 6 methods').

Research on recommendation systems to detect and correct controversial implementation choices typically follows a top-down approach. Recommendations defined at a high level of abstraction are refined into the detection of more concrete symptoms until a straightforward detection strategy is reached. Different recommendation approaches exist that detect issues like design flaws [12] or antipatterns [13]. While these approaches discover similar issues, they often vary significantly in the heuristics, metrics and thresholds they use. These differences have various causes. Heuristics are incomplete by definition. The definition of many metrics remains open to interpretation resulting in different tools that may provide different results for the same metric. And thresholds used tend to be either absolute values that cannot be reused across different applications, or

* Angela Lozano is financed by the CHaQ project of the Flemish IWT funding agency.
** also DCyT - Universidad Nacional de Quilmes and CONICET - Buenos Aires, Argentina

relative values whose cut point may be arbitrary. For these reasons, it is difficult to justify that concrete detection strategies and how they are combined into higher-level recommendations accurately represent all and only those entities that a higher-level recommendation aims to capture.

As opposed to defining high-level recommendations as an ad-hoc combination of lower-level issues, this paper presents a first step towards 'discovering' higher-level recommendations from a detailed analysis of the occurrence of more specific low-level ones. More specifically, our analysis is based on a study and possible interpretation of the *co-occurrence* of low-level recommendations in several applications.

The low-level issues analyzed in this particular paper are the so-called *code critics*. Code critics are a list of detectors for harmful implementation choices in Pharo Smalltalk that signal certain defects or performance issues in Smalltalk source code, mainly in methods and classes. Each critic is defined with a short name and a rationale that explains why that implementation choice could be harmful and, in some cases, also proposes a refactoring. An example of such a code critic is the critic named *'Instance variables not read AND written'* with rationale:

> "Checks that all instance variables are both read *and* written. If an instance variable is only read, you can replace all of the reads with nil, since it couldn't have been assigned a value. If the variable is only written, then we don't need to store the result since we never use it. This check does not work for the data model classes, or other classes which use the `instVarXyz:put:` messages to set instance variables."

Although code critics sometimes report false positives (like the `instVarXyz:put:` messages mentioned in the rationale of the critic above), the Code Critics browser allows one to 'ignore' each reported result individually. Results that have been ignored are saved within the image[1], so that the system remembers that they have been ignored and does not present them again to the developer when the same code critics are checked again later.

Each code critic belongs to one of the following categories: *Unclassified* rules, *Style* issues, *Coding Idiom Violations*, suggestions for *Optimization*, *Design Flaws*, *Potential Bugs*, actual *Bugs* and likely *Spelling* errors. For instance, the code critic named *'Instance variables not read AND written'* is categorized as an *Optimization* issue.

This paper is structured as follows: Section 1 detailed the problem and context of low-level recommendation tools. Section 2 introduces the concept of code critics in more detail, and Section 3 shows how we define the distance function to calculate if code critics co-occur in the analyzed application. Section 4 presents critiques on individual critics and several patters of co-occurring critics. Section 5 concludes our work and presents some future work.

[1] Smalltalk systems store the entire program and its state in an image file.

2 An Introduction of code critics

Although Pharo's *Critic Browser* is designed to be launched by a developer from a menu in the IDE, the tool can also be run programmatically to analyze part of the image with a selected set of critics. In our experiment, we analyzed 120 code critics, 27 applied to classes, and 93 applied to methods. We excluded the category of *Spelling* rules, which check the spelling of comments and identifiers of classes, methods and variables. We are less interested in these rules as they do not refer to either the structure or design of the source code, and tend to generate quite some noise in the results.[2]

Id	Critic name
CC01	A metamodel class does not override a method that it should override
CC02	Class not referenced
CC03	Class variable capitalization
CC04	Defines = but not hash
CC05	Excessive inheritance depth
CC06	Excessive number of methods
CC07	Excessive number of variables
CC08	Instance variables defined in all subclasses
CC09	Instance variables not read AND written
CC10	Method defined in all subclasses, but not in superclass
CC11	No class comment
CC12	Number of addDependent: messages > removeDependent:
CC13	Overrides a 'special' message
CC14	References an abstract class
CC15	Refers to class name instead of 'self class'
CC16	Sends 'questionable' message
CC17	Subclass responsibility not defined
CC18	Variable is only assigned a single literal value
CC19	Variable referenced in only one method and always assigned first
CC20	Variables not referenced

Table 1. Some of the most frequent class-level critics and their identifiers.

Table 1 lists some of the most frequently found class-level code critics, and Table 2 lists some discovered method-level critics. We added an identifier to each of them for easy reference later. For example, CC09 refers to the code critic *'Instance variables not read AND written'*.

For our analysis we studied Moose [5], a Smalltalk platform consisting of a variety of software and data analysis tools. More specifically, we analyzed all packages contained in the downloadable image containing the latest distribution of Moose (i.e., Pharo 1.4). For each package studied (71 in total) we accumulated all critics found in methods and classes, except for those methods and classes

[2] For the same reason they do not even appear in recent versions of the *Critic Browser*.

Id	Critic name
MC01	detect:ifNone: -> anySatisfy:
MC02	Inconsistent method classification
MC03	Law of Demeter
MC04	Methods implemented but not sent
MC05	Rewrite super messages to self messages when both refer to same method
MC06	Sends different super message
MC07	Temporaries read before written
MC08	Unclassified methods
MC09	Uses detect:ifNone: instead of contains:
MC10	Utility methods

Table 2. Some common method-level critics and their identifiers.

related to tests. We excluded the tests because critics about test code often lead to false positives. Test code tends to adhere to other idioms than ordinary code. For instance, test code often contains duplicated code between test methods (due to similar calls to 'assert' or other testing methods). Moreover, test code often contains trial-and-error code to deal with all cases to be tested, which is typically not considered good practice in normal code.

3 Critiques on individual and co-occurring code critics

Our analysis generates two boolean tables per package: one for its classes and another for its methods. Each table shows which source code entities suffer from which critics. Each column represents a method or class of the package, and each row represents which entities are in the result set of a code critic. E.g., suppose we analyze the following class-level code critics in the package Compiler (which is part of the analyzed distribution): *'Instance variables not read AND written' (CC09), 'Sends 'questionable' message' (CC16), 'Excessive number of variables' (CC07), 'Excessive number of methods' (CC06)* and *'Variables not referenced' (CC20)*. Table 3 presents the results[3], where the rows identify the *critiqued* entities for a corresponding critic in the analyzed package. In other words, $critiqued(c, p)$ is a sequence of boolean values $< c(e_1), c(e_2), \ldots, c(e_n) >$ where $c(e_i) = true$ if and only if e_i is the i^{th} entity in package p (by alphabetic order on its fully qualified name), and e_i is in the result set of code critic c.

Next, we calculate the distance between pairs of critics based on the entities they *critique*. The distance between two code critics c_1 and c_2, for a given package p, is calculated by counting the number of classes or methods where the critics do not match (XOR of the critiqued entities), over the number of classes

[3] To limit the size of the example, this table present only a subset of all classes that were critiqued. However, for the sake of the example, in order to illustrate how the approach works, we ask the reader to assume that the classes shown in Table 3 are all the critiqued classes in the package.

	AmbiguousSelector	BlockNode	Encoder	LiteralVariableNode	VariableNode	CommentNode	UndVariableReference	MessageNode	AssignmentNode	ParseNode	MethodNode	Decompiler	Compiler	Parser	BytecodeEncoder	TempVariableNode
CC09	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
CC16	0	1	1	0	1	0	0	1	1	1	1	1	1	0	0	0
CC07	0	1	1	0	0	0	0	1	0	1	1	1	0	1	0	0
CC06	0	1	1	0	0	0	0	1	0	1	1	1	0	1	1	1
CC20	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0

Table 3. Code critics (CC) per class for the Compiler package.

	CC09	CC16	CC07	CC06	CC20
CC09	0	0.81	0.77	0.81	0.83
CC16	0.81	0	0.40	0.50	0.66
CC07	0.77	0.40	0	0.22	0.57
CC06	0.81	0.50	0.22	0	0.66
CC20	0.83	0.66	0.57	0.66	0

Table 4. Distance among the code critics shown in Table 3.

or methods that violate one or both of the critics being analyzed (OR of the critiqued entities). This distance value varies between zero and one. Values close to zero mean that a pair of critics tends to affect the same source code entities.

$$D_p(c_1, c_2) = \frac{|critiqued(c_1, p) \ \oplus \ critiqued(c_2, p)|}{|critiqued(c_1, p) \ \vee \ critiqued(c_2, p)|}$$

For instance, Table 5 calculates the distance between *'Instance variables not read AND written'* and *'Variables not referenced'* based on the presented example. The resulting distance, shown as a shaded cell in Table 4, is 0.83 (i.e., 5/6) because their results differ in five classes, but coincide in one class (BlockNode). Therefore, the critics have low co-occurrence for the results of this package.

CC09: Instance variables not read & written 1111000000000000
CC20: Variables not referenced 0100000101000000
XOR 1011000101000000
OR 1111000101000000

Table 5. Calculation of the distance between a pair of critics based on their results for a given package (shown in table 3).

Using the Boolean table 3 and the distance table 4 we proceed to discard pairs of code critics that do not seem interesting for our analysis, based on three criteria. First, pairs with high distances (greater than 0.9) are discarded as they tend not to co-occur often and therefore are likely to represent accidental matches. Secondly, we discard pairs of critics that *always* occur together (distance zero) in the same source code entities, because they are likely to represent alternative implementations of a same code critic. Thirdly, we exclude all pairs of critics for which one of the code-critics covers more than 90% of all source code entities analyzed, because as a consequence of their high coverage they will show a strong correlation with nearly all other code-critics and thus generate significant noise in the results. In our example, all distances are kept in our analysis. The choice of thresholds of 0.9 and 90% was based on initial experiments where we tried to determine what values would constitute a good cut point to discard less relevant pairs of critics. However, these thresholds should be reevaluated when applying the approach to other code critics, other applications, or different programming languages.

4 Identified patterns

Based on the raw results of our initial analysis, this section presents some interesting critiques which we have observed. Since this is a preliminary research, we do not claim these critiques to be exhaustive nor complete. In the text below, we use the word *critique* to denote the identified patterns in our analysis, and *critic* to refer to Pharo's code critics. We present our critiques as patterns, consisting of a short name, a description and some concrete examples. The patterns are divided in two big categories. The first category describes the critiques discovered by analyzing individual code critics. Note that we limited our analysis of individual code critics to those that appear at least in one of the non-discarded co-occurrences. The second category describes the critiques which stem from the observed correlations between pairs of code critics (extracted from their co-occurrence as explained in Section 3).

4.1 Critiques on Individual Critics

Here we present our critiques on the individual class-level critics of Table 1.

Misleading name. Some code critics have misleading names and should be improved. For example, *'References an abstract class' (CC14)* is misleading. According to the name, a developer could assume that the code critic identifies a class B that is referencing an abstract class A. But in fact it detects the opposite, namely an abstract class A being referred to from somewhere within the analyzed application. A better name would thus be *'Abstract class being referenced'*. The name *'Instance variables not read AND written' (CC09)* is ill chosen too because, looking at how this code critic is implemented, it refers to instance variables which are EITHER read-only, write-only, OR not referenced

at all. A better name for this code critic could therefore be *'Instance variables not fully exploited'*.

Too general. Some critics are too general and could be split into several more specific critics. For example, the critic *'Instance variables not read AND written' (CC09)* mentioned above could be split into three different critics ('unreferenced instance variables', 'only written instance variables', 'only read instance variables'). The critics *'Overrides a special message' (CC13)* and *'Sends 'questionable' message' (CC16)* are about specific messages and could be split into separate critics for each of those messages. This would however lead to many individual critics, but they could be presented as a common group to the user, allowing him to inspect or ignore the details of the individual underlying critics, if he desires to do so.

Too tolerant. We also observed that, despite the fact that some critics seem meaningful and well-defined, they produce mostly false positives. This happens because there are often cases where it is acceptable not to adhere to some critics. However, when a critic produces mainly such false positives, we can wonder whether it is useful to keep the critic. Nevertheless, our results might be biased, since we analyzed only one rather well-designed framework (Moose).

An example of such a critic is *'Refers to class name instead of 'self class' ' (CC15)*, for which we discovered mostly acceptable deviations. For example, in Smalltalk it is quite common and acceptable in methods for checking equality to write `anObject isKindOf: X`, to verify that the type of `anObject` is indeed of a particular class `X` (and not some subclass). Similarly, the expression `self class == X` is often used to check if a given instance of this class is indeed of class X. Another case is when you write `X new`, because you want to be sure to create an instance of X and not of one of its subclasses. A last example is when you write an expression like `X allsubclasses` to refer to the root X of a relevant class hierarchy, and you want to manipulate the individual classes.

Many of the critics which are too tolerant could be refined further in order to avoid catching some of the false positives they produce. For example, if we consider CC15 again, we note that it often regards an expression like `isKindOf: X` used in a method implemented by class X as problematic, but in fact `isKindOf: self class` would be even more problematic, because it would get a different meaning in subclasses. This could be solved by making the critic take into account this case or any other of the above cases as known exceptions to the critic.

Too restrictive. Whereas some critics are too tolerant, others are too restrictive and could miss interesting cases. For example, *'Excessive inheritance depth' (CC5)* uses a threshold of 10 as depth level, but may miss other cases of excessive depth such as classes with inheritance depth 9. Obviously, there is no perfect threshold, but we found 20 additional classes with a depth of at least 9 (as compared to only 10 classes with a depth of at least 10) that should have been reported. We assume the threshold was set high in order to avoid producing too many results, making it harder for the user to process all reported results.

Redundant representation of results. Another source of noise in the results could be the amount of results produced by the critic, even if none of them are false positives. Sometimes, it would suffice to present the results differently to avoid such noise. For example, consider *'Excessive inheritance depth' (CC5)* again. Currently, it reports all leaf classes of hierarchies that suffer from the critic. But this generates many unnecessary results. It suffices to know the root of the hierarchy to start fixing the problem (and additionally, this could allow the user to lower the threshold so that the critic becomes less restrictive too).

Missing critics. Some important critics seem to be missing from the list of code critics. For example, there seem to be little or no critics related to inheritance issues [10], such as *local behavior* in a class with respect to its superclass or subclasses, or good *reuse of superclass behavior and state. Local behavior* identifies methods defined and used in the class that are not overridden in subclasses, often representing internal class behavior, and *Reuse of superclass behavior and state* identifies concrete methods that invoke superclass methods by self or super sends, not redefining behavior of the class. Code critics regarding inheritance could identify bad practices when implementing hierarchies.

Good critics. Whereas in this paper we focused mainly on negative critiques on code critics, we can remark that there are useful and well-designed code critics too. Our ultimate goal is to keep the good critics while identifying those that can be improved, in order to come up with a new and better-structured set of code critics. For example, *'Defines = but not hash' (CC04)* shows all classes that override `=` but not `hash`. If method `hash` is not overridden, then the instances of such classes cannot be used in sets. The implementation of `Set` assumes that equal elements have the same hash code. Another example is *'Method defined in all subclasses, but not in superclass' (CC10)* which detects classes defining a same method in all subclasses, but not as an abstract or default method in the superclass. This critic helps us find similar code that might be occurring in all the subclasses and that should be pulled up into the superclass.

4.2 Patterns of Co-occurring Critics

Now that we have described some critiques based on an analysis of individual code critics, we discuss some critiques derived from our analysis of the co-occurrence of pairs of code critics.

Redundant Critics. Critics are redundant when they detect the same problem. This happens for critics that come in two versions: one which just detects the problem and another one which detects it and at the same time proposes an automated refactoring to the problem. An example of this is *'detect:ifNone: -> anySatisfy:' (MC01)* versus *'Uses detect:ifNone: instead of contains:' (MC09).* Whereas MC01 offers an automated restructuring, in spite of its name MC09 only detects the problem. Although we did discover such cases in our experiment where we ran the critics directly, Pharo's Critic Browser would only use

one of these critics in order to avoid the user to get repeated results. Observe that the solution suggested by critic MC09 differs from the solution proposed by MC01, which can be confusing. Given that a same critic could have several possible refactorings, it would therefore be better to keep refactoring and detection strategies separated, and to have only one detection strategy per critic.

Indirect Correlation. This occurs when the results of two critics overlap significantly, without them having a common root cause. For instance, the following two correlations seem to occur essentially because one of the critics (*CC06*) generates so many results. They are *'Excessive number of methods' (CC06)* vs. *'Excessive number of variables' (CC07)*, and *'Sends 'questionable' message' (CC16)* vs. *'Excessive number of methods' (CC06)*.

Overlap Requires Splitting. A third pattern occurs when two critics produce overlapping results because they have a common root cause. It would be good to split such critics such that the common part becomes one separate critic and the non-overlapping parts become other critics. For instance, *'Instance variables not read AND written' (CC09)* is overlapping with *'Variables not referenced' (CC20)* because both critics detect unreferenced instance variables. While CC09 should be split as explained in section 4.1 (too general), CC20 could be split in a critic for class variables and one for instance variables. The critic for 'unreferenced instance variables' would then become a common subcritic for both CC09 and CC20.

Overlap Requires Merging. This pattern occurs when two code critics that regularly occur together could be combined into a single more specific critic. For instance, in the Smalltalk language, methods are grouped in method protocols representing the purpose of the method. Instance creation methods like `new`, for example, are put in the `'instance-creation'` protocol. The method-level critic *'Inconsistent method classification' (MC02)* is triggered when methods are wrongly classified and *'Unclassified methods' (MC08)* are reported when no protocol was assigned to a method. These critics coincide when an overridden method is unclassified whereas the method it overrides was classified. From the point of view of critic MC02, it is considered as an inconsistent classification since the classification of the parent and child method are different, whereas from the point of view of critic MC08 the child method is unclassified. Combining them in a new dedicated critic *'Inconsistently unclassified methods'* makes sense, because there is an easy refactoring that could be associated to this particular combination of critics, namely to classify the child method in the same protocol as the parent one. For cases where the critics do not overlap, the original critics MC02 and MC08 should still be reported.

Same niche. Sometimes, code critics seem to correlate just because they both refer to a specific kind of source entity. For example, the two independent critics on abstract classes *'References an abstract class' (CC14)* and *'Subclass responsibility not defined' (CC17)* often appear together, simply because they are the

only ones that both apply to abstract classes. (This pattern could be considered as a specific case of Indirect Correlation.)

Almost subset. This pattern occurs when most of the result set for one critic in practice always seems to be a subset of that for another critic. For example, the results for code critic *'Variable referenced in only one method and always assigned first' (CC19)* refers to the same variables reported by *'Instance variables not read AND written' (CC09)*. Indeed, if a variable is used only in one method and always assigned first *(CC19)*, it is likely that this variable will not be read in that same method (or any other method) and thus is reported by *CC09* too.

Ill-defined critic. Correlations between two critics may arise because one of them is ill-defined. If the ill-defined critic were fixed, the correlation would probably disappear. For example, *'Refers to classname instead of self class' (CC15)* correlates with *'Sends 'questionable' message' (CC16)*, because CC15 often gives false positives related to the use of `isKindOf:`, which is also one of the questionable messages. If we would fix CC15 to avoid those false positives, this correlation would likely disappear.

Noisy correlation. This pattern describes critics that seem to be correlated to many other critics and therefore produce too much noise. They could better be removed if the overlap with another critic is not strong (likely to be only accidental matches). For example, *'Excessive number of Methods' (CC6)* has this problem, because the more methods a class has, the higher the chance that the class suffers from other critics as well.

High-level critics. Whereas in this section we analyzed the co-occurrence of critics mainly by focusing on their shortcomings, in forthcoming research we will analyze the results more in-depth and will also identify good, desired or expected correlations between critics. For example, the correlation between *'Utility methods' (MC10)* and *'Law of Demeter' (MC03)* is not unexpected as it may indicate an imperative (non object-oriented) programming style.

5 Discussion, Conclusion and Future Work

This paper presented our initial results of an analysis of low-level code critics detected on the Moose system, a large and mature Smalltalk application. The results of this analysis can help us identify which low-level critics could benefit from redefinition or refactoring so that they would provide more accurate or meaningful results, as well as how to combine them into more *high-level critics* to improve the recommendations they provide.

As future work, we plan to provide a more in-depth analysis, including a deeper analysis of the method-level critics, and propose concrete improvements, combinations and refactorings of the existing code critics. This analysis could then be repeated iteratively, to further improve the improved critics, again by

analyzing their correlations, until we eventually reach a stable group of proposed critics.

Finally, although in this paper we focused on Pharo Smalltalk's code critics only, we believe the ideas and approach presented in this paper to be easily generalizable to other code checking tools and programming languages. To confirm this, we have started to analyze other code checking tools for similar correlations and improvements: CheckStyle [2], PMD [7] and FindBugs [4, 11] for Java, Splint [9] or Cppcheck [3] for C, Pylint [8] for Python, FxCop for .NET, PHP Mess Detector [6] for PHP and Android Lint [1] for Android programming. For each of these tools, we performed an initial analysis on a single application. We observed that, in spite of the fact that some of these tools focus on checks that are quite different from Pharo's code critics, our approach could still be used to analyze those tools. Whereas for most tools we indeed found many examples similar to the critique patterns mentioned in this paper, for some tools we discovered only very few correlations. This could be due to the particular applications that were analyzed (indeed, in our analysis of the 51 packages of Moose too, there were a few packages that did not have many critics). Or it could suggest that, while the approach remains applicable, it may be less relevant for some of the tools we analyzed. This may for example be the case for tools that are already quite mature and offer a stable and orthogonal set of checks. More experiments are needed to confirm this. This may be the topic of a forthcoming paper.

References

1. Androidlint. `http://tools.android.com/tips/lint`. Accessed: 2015-03-30.
2. Checkstyle. `http://checkstyle.sourceforge.net`. Accessed: 2015-03-30.
3. Cppcheck. `http://cppcheck.sourceforge.net/`. Accessed: 2015-03-30.
4. Findbugs. `http://findbugs.sourceforge.net`. Accessed: 2015-03-30.
5. MOOSE. `http://www.moosetechnology.org/`. Accessed: 2015-03-30.
6. Phpmd. `http://phpmd.org/`. Accessed: 2015-03-30.
7. PMD. `http://pmd.sourceforge.net/`. Accessed: 2015-03-30.
8. Pylint. `http://www.pylint.org/`. Accessed: 2015-03-30.
9. Splint. `http://www.splint.org/`. Accessed: 2015-03-30.
10. G. Arévalo, S. Ducasse, S. Gordillo, and O. Nierstrasz. Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis. *Inf. Softw. Technol.*, 52(11):1167–1187, Nov. 2010.
11. D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA 2004*, pages 132–136. ACM, 2004.
12. R. Marinescu. Detecting design flaws via metrics in object oriented systems. In *Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 173–182. 2001.
13. N. Moha, Y.-G. Gueheneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *Proc. of the Int'l Conf. on Automated Software Engineering (ASE)*, pages 297–300. IEEE Computer Society, 2006.