Detecting Design Rule Violations

Sunny Wong and Yuanfang Cai
Drexel University
Philadelphia, PA, USA
{sunny, yfcai}@cs.drexel.edu

Miryung Kim
The University of Texas at Austin
Austin, TX, USA
miryung@ece.utexas.edu

Michael Dalton
Drexel University
Philadelphia, PA, USA
mcd45@cs.drexel.edu

ABSTRACT

In this paper, we present an approach to detect *design rule violations* that could cause software defects, modularity decay, or expensive refactorings. Our approach is to compute the discrepancies between how components *should* change together based on the modular structure framed by design rules, and how components *actually changed* together revealed by how modification requests were fulfilled. Our contributions include a design violation detection framework and a design-rule based impact scope prediction algorithm. We evaluated our approach using the version history of three large-scale open source software projects. We examined all identified violations to check whether they were refactored or recognized by the developers in later versions.

Our results show that (1) on average 73% of the violations we identified were either recognized or refactored in later releases (when using .5 confidence and varying support from 2 to 10 in Hadoop); (2) our approach can identify problematic design violations much earlier than manual identification by developers; and (3) the identified violations cover multiple *bad smells*, such as tight coupling and code clone.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance and Enhancement—*refactoring, restructuring*; D.2.10 [**Software Engineering**]: Design—*design rule violation, refactoring*

Keywords

design rule violation detection, refactoring, bad code smells

1. INTRODUCTION

Empirical studies have revealed strong correlations between software defects and unintended violations of design decisions [7, 20]. Some design violations are not easily detectable by traditional software verification and validation techniques because they may not influence the functionality of software systems directly. For example, inexperienced developers may forget to remove experimental scaffolding code that should not be kept in the final product, and an application programming interface (API) may be accidentally defined using non-API classes [14]. Such design violations could cause modularity decay over time and may require expensive system-wide refactorings.

In this paper, we present an approach to detect and locate design violations early in the development process. This approach automatically detects design violations by comparing inferred design rules [1]—which components *should* change together according to the modular structure of software—with inferred change coupling—which components *actually* change together according to revision histories.

Consider the following example. If class `B` implements interface `A`, then `A` can be seen as a *design rule* that is designed to be stable, and `A` should not be affected by changes to its subordinate design decisions, such as `B`. If `A` always changes due to `B`'s modifications, this may cause ripple-effects to other classes implementing `A`. If the actual code change history shows that changing `B` always requires changing `A`, it implies that either the reality deviates from the designer's assumption or suspicious dependencies have been introduced through quick and dirty maintenance activities. In these cases, the modular structure framed by the original design rules is not respected and we call such a symptom as a *design rule violation*, or *design violation* for short.

We applied our approach to the version histories of three large-scale open source software systems: 15 releases of Hadoop Common,[1] 9 releases of Derby,[2] and 11 pre-releases of JEdit.[3] If an identified violation was problematic, it is possible that developers recognized this problem and fixed it in a later release through a refactoring. We consider an identified violation as a *confirmed violation* if it was indeed addressed by developers later. We used two complementary evaluation methods. First, we compared our design violations with refactorings automatically reconstructed using Kim et al.'s API matching technique [16]. Second, for the remaining violations, we manually examined modification requests to see whether those violations were later refactored or at least recognized. Since it is possible that the design rule violations we detected in recent versions are not addressed or even realized by the developers yet, we also manually examined the corresponding code to see whether they embody obvious design problems such as cyclical dependencies.

Submitted to the 18th International Symposium on the Foundations of Software Engineering (FSE).

[1] http://hadoop.apache.org/common/
[2] http://db.apache.org/derby/
[3] http://www.jedit.org/

The evaluation shows that our approach can accurately identify design violations. For example, with .5 confidence and varying support from 2 to 10, on average, 73% percent of the violations we identified were confirmed violations. However, the accuracy of our approach depends on the number of modification requests to be used as input. In fact, the more modification requests are available, the more accurate our approach is. For example, with 344 modification requests in Hadoop, the precision of identified violations ranges from 25% to 100%, depending on various support and confidence levels, while there were much fewer confirmed violations for Derby and JEdit with much less precision (ranging from 10% to 75%) because of the dearth of available modification requests that could be mapped to change-set solutions. Second, our approach identifies design violations much earlier than manual identification by developers, showing that our approach has the potential to reduce maintenance costs by identifying refactoring opportunities early in the development process. Third, the identified violations include various types of *bad smells*, such as tight coupling of components and cloned code.

The rest of this paper is organized as follows. Section 2 presents related work and how our approach differs from existing approaches. Section 3 describes our design violation detection approach and several background concepts. Section 4 details our evaluation method and empirical results. Section 5 discusses the strengths and limitations of our approach and Section 6 concludes.

2. RELATED WORK

In this section, we compare and contrast our approach with other related research topics.

Dependency Structure and Software Defects.

The relation between software dependency structure and defects has been widely studied. Many empirical evaluations (e.g., Selby and Basili [20], Cataldo et al. [7]) have found that modules with lower coupling are less likely to contain defects than those with higher coupling, and various metrics have been proposed (e.g., Chidamber and Kemerer [8]) to measure coupling and failure proneness of components. The relation between change coupling [11] and defects has also been recently studied. Cataldo et al.'s [7] study revealed strong correlation between density of change coupling and failure proneness. Fluri et al.'s [9] study shows that a large number of change coupling relationships are not entailed by structural dependencies. The purpose of these studies are to statistically account for the relationship between software defects and logical and syntactic dependencies. In contrast, our approach locates the concrete files involved in problematic design violations that may cause modularity decay and software defects.

Automatic Detection of Bad Code Smells.

Fowler [10] describes the concept of *bad smell* as a heuristic for identifying redesign and refactoring opportunities. Example bad smells include code clone and feature envy (when a class excessively uses methods of another class). Garcia et al. [12] proposed several architecture-level bad smells. To automate the identification of bad smells, Moha et al. [17] presented the Decor tool and domain specific language (DSL) to automate the construction of design defect detection algorithms. For example, to detect spaghetti code, they use the DSL to describe code that contains long methods, methods with no parameters, no inheritance, the use of global variables, and no polymorphism. Based on such description, their framework generates an algorithm that can identify code fragments with bad smells. Several other techniques [21–23] automatically identify bad smells that indicate needs of refactorings. For example, Tsantalis and Chatzigeorgiou's technique [22] identifies *extract method* refactoring opportunities using static slicing. Detection of some specific bad smells such as code duplication has also been extensively researched. Higo et al. [13] proposed the Aries tool to identify possible refactoring candidates based on the number of assigned variables, the number of referred variables, and dispersion in the class hierarchy. A refactoring can be suggested if the metrics for the clones satisfy certain predefined values.

Our design rule violation detection approach is different in several aspects. First, our approach is not confined to particular types of bad smells. Instead, we observe that many types of bad smells are instances of design rule violations that can be uniformly detected by our approach. For example, when code clones change frequently together, our approach will detect that this co-change pattern deviates from design-rule based impact scope prediction. Second, the number of bad smells detected by these approaches can be very large, covering many parts of the system, making it hard to determine which part should be refactored first. In contrast, our approach detects violations that happened most recently and frequently, having the potential to identify the most prominent problems that are worthwhile to address soon. Similar to our approach, Ratzinger et al. [18] also detect bad smells by examining change couplings derived from revision histories. However, their approach leaves it to developers to identify potential design violations from change coupling history visualization. Third, our approach uses a high-level design model representation (augmented constraint network [4, 5]) as a basis for describing design decisions. Though, we used design models automatically derived from source code in this paper, this approach can be used with developer-provided, high-level design models.

Impact Scope Prediction.

Various approaches (e.g., Bohner and Arnold [3], Ying et al. [28], Zimmermann et al. [29]) have been proposed to predict the impact scope of a modification. Cai and Sullivan's ACN-based change impact analysis [6] uses constraint solving to list *all* minimally differing designs, given a specific design decision change. The purpose of these previous approaches are to increase prediction accuracy so that the designers can estimate modification cost. In contrast, our approach focuses on identifying design violations that are evidenced by co-change patterns.

Refactoring Reconstruction.

Refactoring reconstruction (RR) techniques [16, 24] compare two program versions and look for a predefined set of refactoring patterns: move a method, rename a class, add an input parameter, etc. (Comparison between existing RR tools is described elsewhere [27].) While program differencing tools stop at mapping corresponding code elements, RR tools infer refactorings that explain the identified correspondences at the level of a method, class, or file. While these approaches retrospectively examine what refactoring activ-

ities happened in the past, our approach detects potential design violations, thus suggesting which part of system may require refactorings. In this paper, we leverage an automatic refactoring reconstruction tool [16] to evaluate our design rule violation detection approach.

3. DESIGN RULE VIOLATION DETECTION

In this section, we describe a motivating example and an overview of our design rule violation detection approach. We also describe how we predict impact scope based on inferred design rules and how we predict impact scope based on change coupling.

3.1 Motivating Example

Alice is a project manager. Her team just released version n of a software system. A lot of modification requests (MRs) were fulfilled during this release, including both bug fixes and feature enhancements. These modifications were accomplished by multiple developers with different levels of capability and experience. Before Alice launches the development process for release $n+1$, she wants to make sure that the modular structure of the system is well-maintained. That is, she would like to check that there was no quick and dirty implementation, which introduced unexpected dependencies or broke important design rules. Fixing these problems would ensure that they do not accumulate into severe modularity decay. Our design violation detection framework, *Clio*[4], analyzes software configuration management (SCM) history and modification requests to help Alice identify potential design problems.

As a design problem reported to Alice, consider the example shown in Figure 1. Class `Grocery` is a subclass of `Item`. The public interface of class `Item` is a design rule [1] in the sense that it hides changes to the implementation of `Item` and it should remain stable. Using *Clio*, design-rule based prediction would never report that changes to the implementation of `Grocery` will require compensating changes to its superclass `Item`. However, if change-coupling based prediction shows that `Item` always changes due to changes in `Grocery`, then a set of discrepancies will be reported to Alice showing that there is a deviation involving `Grocery` and `Item` and with high support and confidence. Such deviation is problematic because it may cause a ripple effect to other components that use `Item`'s interface. We suspect that changes to `Grocery` is violating `Item`'s design rule, meaning that there exists a problematic dependency between them.

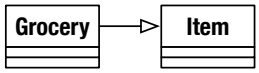

Figure 1: Example UML Diagram

3.2 Framework Overview

The design violation detection framework takes the following artifacts as input: (1) the original design structure of the software before implementing these modification requests, that is, the design model of version $n-1$, (2) the revision history of the project, and (3) the information about which and how modification requests in the last release were fulfilled. We call the first set of files that need to be changed to fulfill a modification request, the change source, for which the change scope is to be identified. A change source can be identified by examining the MR's descriptions, comments, etc. For each resolved MR, we obtain a set of files that were changed to implement the MR, which we call as a *solution* change-set S. Numerous MR tracking systems can be integrated with version control systems, making it easier to associate MRs with their corresponding change sets.[5]

Figure 2 shows an overview of our design violation detection approach. *Moka* [25] is our reverse-engineering tool that derives a UML class diagrams from compiled Java binaries. The *uml2acn* [25,26] is our tool that infers design structure from a UML class diagram and transforms it into an *augmented constraint network* (ACN) [4,5], a design model that formalizes the key concepts of Baldwin and Clarks's design rule theory [1]. Minos is our ACN-based design construction and modularity analysis tool [26]. Minos employs a plug-in architecture that facilitates the addition of new analyses.

Given a change source, we predict its impact scope using two different prediction algorithms: (1) *dr-predict*, an algorithm based on design-rules derived from an ACN (2) *logic-predict*, an algorithm based on co-change patterns (i.e., change coupling). The *dr-predict* plug-in outputs a set of files that *are likely* to be changed together with the specified change source according to the design rule theory. Section 3.3 presents the algorithm of the *dr-predict*. The output of *dr-predict* is denoted by FileSet A in Figure 2.

Given the revision history, the *extract* plugin of *Clio* computes change coupling. Given a change source, the *logic-predict* plugin of *Clio* reports how components were *actually changed* together, recorded in FileSet B. Using the impact scopes A (from the ACN model) and B (from change coupling analysis), and a MR solution S, the *detect* plug-in computes a set of discrepancies, called the *discrepancy set* $D = (B \cap S) \backslash A$. By using $B \cap S$, we identify the subset of the solution that is logically coupled with the change source and only consider the files that change frequently with the change source.

3.3 Design Rule-based Scope Prediction

This section discusses our design-rule based impact scope prediction algorithm that is based on our prior work on the augmented constraint network (ACN). We first describe the key concepts of the ACN and design rule hierarchy. Then we describe our new impact scope prediction approach based on the ACN.

Background: Augmented Constraint Network.

The *augmented constraint network* (ACN) is a design model created by Cai and Sullivan [4,5]. This model is designed to express the assumption relations among design decisions. Figure 3 shows a sample ACN model automatically derived from the UML class diagram shown in Figure 1. An ACN consists of a *constraint network* that models design decisions and their relations, a *dominance relation* that formalizes design rules, and a *cluster set* in which each cluster represents a different way to partition a design. A constraint network consists of a set of design *variables*, which model design di-

[4]Clio is the Greek muse of history.

[5]Consistent with the definitions of Ying et al. [28], a *modification request* can be either a bug fix or enhancement task. A *solution* (or a change-set) is a set of of files that contribute to an implementation of a modification request.

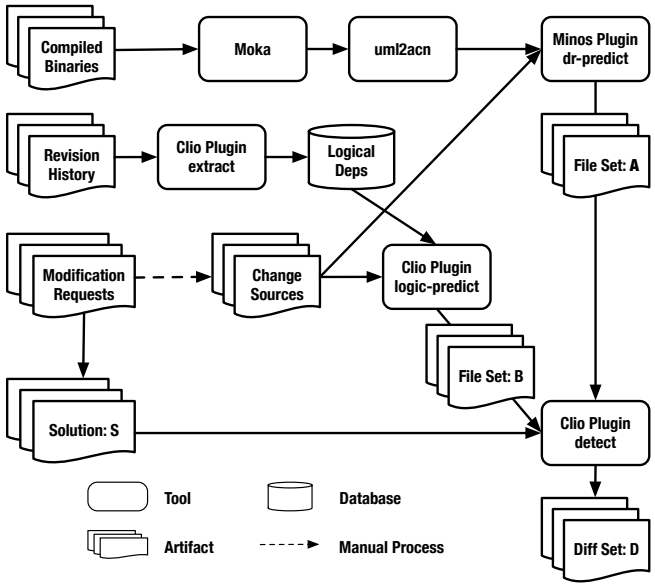

Figure 2: Approach Overview

Variables:

$$\begin{aligned} Grocery\_interface &: \{orig, other\} \\ Grocery\_impl &: \{orig, other\} \\ Item\_interface &: \{orig, other\} \\ Item\_impl &: \{orig, other\} \end{aligned}$$

Constraints:

$$\begin{aligned} Grocery\_impl = orig &\Rightarrow Grocery\_interface = orig \\ Item\_impl = orig &\Rightarrow Item\_interface = orig \\ Grocery\_impl = orig &\Rightarrow Item\_interface = orig \\ Grocery\_interface = orig &\Rightarrow Item\_interface = orig \\ Grocery\_impl = orig &\Rightarrow Item\_impl = orig \end{aligned}$$

Dominance Relation:

$$\begin{gathered} (Grocery\_impl, Grocery\_interface) \\ (Item\_impl, Item\_interface) \\ (Grocery\_impl, Item\_interface) \\ (Grocery\_interface, Item\_interface) \\ (Grocery\_impl, Item\_impl) \end{gathered}$$

Figure 3: Augmented Constraint Network

mensions, environment conditions, and their domains; and a set of logical *constraints*, which model the relations among variables. The main relations we model using an ACN is the *assumption* relations among design variables. For example, the third constraint in Figure 3 models that the implementation of class `Grocery` assumes that the interface of class `Item` is as originally agreed.

Baldwin and Clark define *design rules* as stable design decisions that dominates other subordinating decisions and decouples other wise coupled decisions. We augment the constraint network with a binary *dominance relation* to model asymmetric dependence relations among decisions, formalizing the concept of design rules. For example, the third dominance relation pair in Figure 3 indicates that the decision for how to implement class `Grocery` cannot influence the design of `Item`'s interface; in other words, we cannot arbitrarily change the interface of `Item` to simplify class `Grocery`'s implementation because other components may rely on that interface. Our design violation detection approach aims to identify situations where such a design rule is not followed.

Impact Scope Prediction.

Our algorithm for predicting impact scope from an ACN leverages Robillard's software dependency analysis algorithm [19] and our previous work on design rule hierarchies [26]. Robillard presented an algorithm for recommending relevant code from initial elements of interest (change source) by analyzing the topology of a software dependency graph. We apply a variation of his algorithm using a design rule hierarchy graph as the underlying dependency structure.

In our previous work [26], we showed that a directed-acyclic graph (called the DR hierarchy) can be automatically derived from an ACN, revealing the impact scope of each design rule. This graph provides two useful properties that can be leveraged in our impact scope prediction algorithm. First, the DR hierarchy explicitly identifies the design rules for any part of the design. With this information, we can ensure that design rules are not predicted as being in the impact scope of the change source because the design rules should remain stable. Second, each vertex of the DR hierarchy aggregates a set of ACN variables, approximating an independent task assignment; in other words, each vertex contains a set of design decisions that can and should be made together, and vertices in the same hierarchy level can be assigned as tasks to be completed in parallel. Edges represent a potential pairwise dependence relation—such that if there is an edge $u \rightarrow v$ then a change to u may require a change to v. Because of the property of independent tasks, we assume that if one ACN variable in a vertex is in our impact scope, then all ACN variables in that vertex are in also our impact scope.

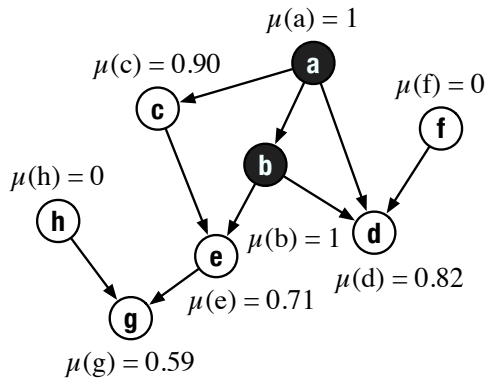

Figure 4: Example DR Hierarchy Graph

Figure 4 shows an example DR hierarchy graph, which we use to illustrate our design-rule based impact scope prediction algorithm. The vertices with shaded background and white text are the change source extracted from the mod-

ification request. Starting from these vertices, we assign a weight μ, in the range $[0, 1]$, to each vertex, in a breadth-first order. The change source vertices are assigned the maximum weight of 1 and added to a set S, which we call the set of interest. From vertex a, we examine its neighbors and assign them a weight, because these are the elements, identified by the ACN and DR hierarchy, as being subordinate design decisions to the design rule a, and potentially need to change if a changes. While traversing the graph to assign weights, we ignore those vertices that are design rules to the change source. Since the DR hierarchy identifies the design rules of a system, we can easily determine which vertices to ignore.

Robillard [19] defines a formula for computing the weight of a vertex:

$$\mu_0 = \left(\frac{1 + |S_{forward} \cap S|}{|S_{forward}|} \cdot \frac{|S_{backward} \cap S|}{|S_{backward}|} \right)^{\alpha}$$

Consider computing the weight of c. Here $S_{forward} = \{b, c, d\}$ is the set of neighbors, through outgoing edges, of a and $S_{backward} = \{a\}$ is the set of neighbors, through incoming edges, of c. The value of μ_0 is scaled by a constant α (defined to by 0.25 in both Robillard's work and our evaluation), however the value of this constant does not affect the order of suggested elements. Using this formula, we compute the weight of c as $\mu(c) = \mu_0(c) \cdot \mu(a) = (\frac{1+1}{3} \cdot \frac{1}{1})^{0.25} \approx 0.9$. Repeating this, we can assign weights to all the neighbors of a. Then we compute the weights for the neighbors of b. In computing the weight of e, we have $S_{forward} = \{d, e\}$ and $S_{backward} = \{b, c\}$. So that $\mu(e) = \mu_0(e) + \mu(b) = (\frac{1+0}{2} \cdot \frac{1}{2})^{0.25} \approx 0.71$. Since we already assigned a weight to the other neighbor of b, through a, we are finished with one iteration of our algorithm. Basically, by using Robillard's formula, we assign a higher weight to vertices that share more edges with elements in the set of interest S.

To start the next iteration of our algorithm, we take all the vertices that have just been assigned weights, add them to the set of interest S, and use them as the starting points for weight assignment. For example, since e was assigned a weight in the previous iteration, we next compute the weights of its neighbors, which is only g. So $S = \{a, b, c, d, e\}$, $S_{forward} = \{g\}$, and $S_{backward} = \{e, h\}$, giving us a weight for g of $\mu(g) = \mu_0(g) \cdot \mu(e) = (\frac{1+0}{1} \cdot \frac{1}{2})^{0.25} \cdot (\frac{1}{4})^{0.25} \approx 0.59$. In scaling the weight of g by the weight of e, we are basically assigning lower weights to vertices that are further away from change source.

We repeat this process of iteratively assigning weights to vertices until the new weights fall below a certain threshold. All vertices that were not assigned a weight are considered to have the minimum weight of 0. Figure 4 shows the weights for each vertex after all weights are assigned. The vertices whose weights are above the threshold are then recommended as being in the impact scope. Below, we discuss how this minimum threshold is determined.

3.4 Change Coupling-based Impact Scope Prediction

The *extract* plug-in of *Clio* takes source code revision history as input and extracts the *change coupling* between files, storing the support and confidence values between files into a database for later use. The *logic-predict* plug-in implements the change coupling analysis algorithm created in previous research by Ying et al. and Zimmermann et al. [28, 29].[6] It reads the change couplings from the database and predicts the impact scope (noted as B in Figure 2) from the change source. A file is predicted to be in the impact scope if the corresponding co-change pattern's support and confidence are above the minimum support and confidence thresholds.

3.5 Discrepancy Analysis

When using *dr-predict* and *logic-predict* plug-ins to calculate design-rule based and history-based impact scopes, we vary the heuristic thresholds to find the values that maximize the average F_1 score over all the MRs. With *dr-predict*, we vary the minimum weight threshold from 0 to 0.95 (by increments of .05). With *logic-predict*, we vary minimum support from 2 to 10 and the minimum confidence value from 0 to 0.95 (by increments of .05). and Once we have *discrepancy sets*, we use the heuristics of support and confidence for determining which files in the *discrepancy sets* are potential *violations* and deserve further investigation. A violation is defined as a set of files that our approach recommends to the maintainer. As an example of determining a violation, we consider two MRs with the same change source of a. Suppose that the discrepancy set is {{a,b,c}, {a,b}}. Then, we say that {a,b} is involved in a potential violation with support 2 and confidence 1. On the other hand, {a,b,c} is involved in a discrepancy with support 1 and confidence 0.5. In our evaluation, we considered various support and confidence values and their impact on identifying actual design problems.

4. EVALUATION

We explore the following questions to assess the effectiveness of our design rule violation detection approach.

Q1. How accurate are the design violations identified by our approach? In order to measure the precision of identified design violations, we examine the project's version history to see whether and how many violations we identified in earlier versions are refactored in later versions or recognized as design problems by developers. This is a conservative assessment because it is possible that some violations we identified have not been recognized by the developers, and could be refactored in future releases. We do not calculate the *recall* of our result because it is not possible to find all possible design issues in a system.

Q2. How early can our approach enable designers to identify problematic violations in the development life-cycle? To answer this question, for each confirmed violation, where it was actually refactored or recognized by the developers

Q3. What are the characteristics of design violations identified by our approach? We manually examine the types of design violations found by our approach to see whether they have symptoms of *bad code smells*, such as cyclic dependencies, code clones, poorly designed inheritance hierarchy, etc.

[6] Consistent with Zimmermann et al. [29], the frequency of a set in a set of transactions T is $frq(T, x) = \{t | t \in T, x \subseteq t\}|$. The support of a rule, $x_1 \Rightarrow x_2$, by a set of transactions T is $supp(T, x_1 \Rightarrow x_2) = frq(T, x_1 \cup x_2)$. The confidence of a rule is $conf(T, x_1 \Rightarrow x_2) = \frac{frq(T, x_1 \cup x_2)}{frq(T, x_1)}$.

Table 1: Characteristics of Subject Programs

Subjects	*Hadoop*	*Derby*	*JEdit*
SLOC	13K to 64K	313K to 360K	64K to 98K
# Revisions	3001	2732	4183
# Release Pair	15	9	11
# MRs	344	144	37
# *Discrepancy Sets*	313	144	37

Table 2: Identified Design Violations (support 1, confidence 0)

	$\|V\|$	$\|V \cap R\|$	$\|V \cap M\|$	$\|CV\|$	Pr.
Hadoop	825	91	117	208	0.25
Derby	316	0	39	39	0.12
JEdit	29	0	3	3	0.10

4.1 Subjects

We choose the open source Hadoop, Derby, and JEdit projects as our evaluation subjects. Both Hadoop and Derby employ an effective bug tracking system, JIRA,[7] which is integrated with their SCM tools. Because of this integration, we can easily associate the change-set solutions to MRs. JEdit uses a simpler bug tracking system that is not integrated with their SCM tool. However, since 2007, developers have been manually posting comments to MRs with associated SCM transactions. Table 1 characterizes, for each subject, the number of lines of size, the number of revisions, the number of pairs of releases examined, the number of modification requests used as input, and the number of discrepancy sets (also called diff sets) generated by our approach.

Hadoop is a Java-based system for distributed computing in a map-reduce paradigm. Since Hadoop consists of several sub-projects, we focused on *Hadoop Common*, formerly called *Hadoop Core*. We applied our approach to the first 15 releases, 0.1.0 to 0.15.0, covering a time period of about three years of development. We used a total of 344 modification requests as input to our approach.

Derby is a relational database system, implemented in Java. Derby was originally a commercial product, developed by IBM and released to the open source community—becoming an Apache project in late 2004. We used 9 releases of Derby, from release 10.1.1.0, to 10.5.3.0. These releases covered almost four years of development. Although Derby also employs JIRA, many modification requests did not have corresponding change-set solutions. Hence, only 144 were available for our evaluation. Note that although Derby is about 6 times larger than Hadoop, the number of available MRs is less than half of Hadoop.

JEdit is a Java-based text editor, with features such as syntax highlighting and the ability to install additional plug-ins. Although there are numerous plug-ins for JEdit, we only focused on the main application for our evaluation. We used 11 pre-releases of JEdit, from 4.3pre7 to 4.3pre18, because these were the only versions where we could find MRs that can be mapped to change-set solutions. Since the bug tracking system was not integrated with their SCM tool, only 37 MRs were available for our evaluation. Compare with Hadoop, the latest version of JEdit is about 1.5 larger than the latest version of Hadoop, but only about 1/10 of the number of MRs as Hadoop are available for our evaluation.

4.2 Evaluation Procedure

To assess the quality of reported violations, we checked whether developers resolved the violations in later versions through refactoring or redesign of the system. First, we compared the detected design rule violations with refactorings that were automatically found by Kim et al.'s API matching tool [16]. This API matching tool takes two program versions as input and detects nine different types of refactorings at a method-header level. A method-header is defined as a tuple, (`package:String`, `class:String`, `procedure:String`, `input_argument_list:List[String]`, `return_type:String`). Each API-level refactoring is a tuple modifying operation. This refactoring reconstruction algorithm extracts method-headers from both old and new versions respectively, finds a set of seed matches based on name similarity, generates candidate high-level transformations based on the seed matches, and iteratively selects the most likely high-level transformation to find a set of method-header level refactorings. Further detail on transformation semantics and refactoring reconstruction is described elsewhere [15]. We found 1818 refactorings in the first 20 releases of Hadoop, 1297 refactorings in 9 releases of Derby, and 77 refactorings in 18 pre-releases of JEdit.

Like many refactoring reconstruction techniques, this technique suffers from two limitations: (1) found refactorings are not necessarily correct API-level refactorings (the issue of precision) and (2) not all refactorings are detected by the technique (the issue of recall). However, we believe that this automated technique is still useful for assessing design rule violations because this technique has a 5.01% higher precision than existing method-header level refactoring reconstruction techniques according our recent comparative study [27], and there exists no technique that can automatically identify all different types of refactorings by comparing two program versions.

As these automatically reconstructed refactorings are method-header level refactorings, we aggregated the refactorings up to a class level to compare with violations that are reported at a file level. Then we checked whether each violation overlaps with any class-level refactorings. If the intersection between two sets are not an empty set, we say that the reported violation is confirmed to be a true design violation as they were later refactored by developers. For each violation that is matched with a reconstructed refactoring, we manually checked the refactoring to verify that it was indeed a correct refactoring that fixes design problems since the API-matching tool can report false positive refactorings. Furthermore, to complement this automated validation approach, we also manually inspected modification request descriptions and change logs in the version history to check whether programmers fixed, or at least plan to fix, those design violations through redesign or refactoring activities. We also check corresponding code to see whether they embody obvious design problems such as cyclical dependencies.

4.3 Results

We ran our experiments on a 2.53Ghz Intel Core 2 Duo MacBook Pro with 4GB of RAM. Running our prediction algorithm on the test subjects took about one minute per release.

[7] http://www.atlassian.com/software/jira/

Accuracy of Identified Design Violations.

We first address evaluation question *Q1*. Table 2 shows the maximal number of violations that could be identified from the subject projects by our approach, using the minimum support threshold 1 and confidence threshold 0. Although we do not advocate using these threshold values, they serve to show the upper-bound on the number of violations our approach can produce, and the lower-bound on the precision of our approach. Table 2 shows the total number of violations identified by our approach ($|V|$), the total number of violations that match with automatically reconstructed refactorings ($|V \cap R|$), the total number of remaining violations that were confirmed to be true violations based on manual inspection ($|V \cap M|$), the total number of *confirmed violations* $|CV|$ (which is $|V \cap R| + |V \cap M|$), and *precision*, which is defined as the number of confirmed violations out of the total number of reported violations: $\frac{|CV|}{|V|}$. The table shows that, with minimal support and confidence thresholds, our approach identified 825 discrepancies in Hadoop, out of which 208 were confirmed as violations. Among the 208 confirmed violations, 91 were confirmed through matching with reconstructed refactorings and 117 were confirmed through manual inspection of MRs.

We see from Figure 5 that as we increase the minimum support and confidence thresholds for deriving violations from discrepancy sets, the more accurate our predictions become. For example, with a minimum support of 4 and minimum confidence of 0.15, we identified 94 violations in Hadoop, of which 68 were confirmed, producing a precision of 0.72. Out of these 68, 30 were confirmed by reconstructed refactorings and 38 were confirmed by manual inspection of MRs. With a support of 8 and a confidence of 0.3, we obtain 0.8 precision. Increasing the support threshold to 10, we achieve a 0.83 precision. Alternatively increasing the confidence of 0.6, we get 1.0 precision. However, as we see from Figure 6, increasing the support and confidence values also reduces the number of violations identified. Hence, selecting a support and confidence pair to use presents a trade off between having a lower false positive rate (higher precision) and a lower false negative rate (more violations identified).

Due to the much smaller number of modification requests available for Derby, discrepancy set patterns did not recur enough for us to obtain violations with higher minimum support value than 2. With a minimum support of 2 and confidence of 0, we found 36 violations, none of which were matched with automatically reconstructed API refactorings. We manually confirmed 21 of them as true violations based on later MR descriptions and code inspection, producing a precision of 0.58. Increasing the minimum confidence improved our precision. With a confidence of 0.95, we found 16 violations, of which 12 of them were confirmed, producing a precision of 0.75.

With even fewer number of modification requests, our tool identified discrepancy sets for JEdit but there were no recurring patterns among these discrepancy sets to produce any violations. For our evaluation, we considered these discrepancy sets as violations with minimum support of 1 and computed the precision from varying confidence from 0 to 0.95. With confidence of 0, we found 29 violations, of which 3 were confirmed, giving us a precision of 0.10. Increasing the minimum confidence confidence improved our precision. With a confidence of 0.95, we found 3 confirmed violations out of 20 violations, giving us a precision of 0.15. In Section 5, we further elaborate on the reasons why we obtained such few violations for these subjects.

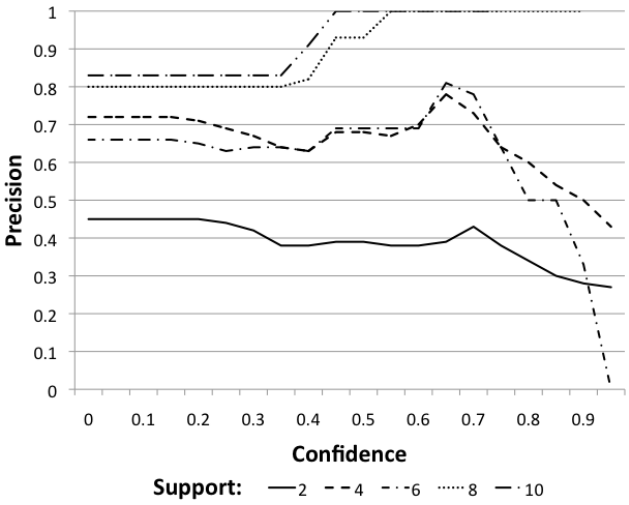

Figure 5: Precision (Hadoop)

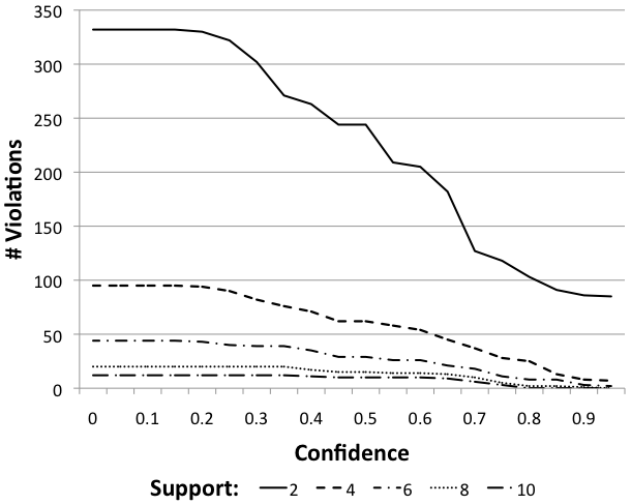

Figure 6: Number of Identified Violations (Hadoop)

In-depth Case Study: Hadoop.

We conducted an in-depth study of violations found in Hadoop. Figure 6 shows the number of identified violations for various minimum support and confidence thresholds. We categorized them into three categories: (1) violations that are verified by automatically reconstructed refactorings, (2) violations that we manually verified based on MR descriptions but not reconstructed refactorings, or (3) false positive violations. In addition, we also investigated (4) refactorings that are not identified as design violations in earlier versions (potential false negatives).

1. Violations that match automatically reconstructed refactorings: Our tool found 76 violations belonging to this category (support 2, confidence 0). For example, in release 0.3.0, our tool identified a violation involving `FSDirectory` and `FSNamesystem`. `FSNamesystem` was the only client

of the `FSDirectory.isValidBlock` method and this method created a tight coupling between the two classes. An API-level refactoring was identified in release 0.13.0, showing that the `isValidBlock` method was moved from `FSDirectory` to `FSNamesystem`. Upon further investigation, we saw that, in the subsequent release, the method was made *private*. In this case, our tool identified this *move method* refactoring opportunity, 11 releases prior to the actual refactoring.

2. Violations that were manually confirmed: Our tool found 75 violations belonging to this category when using support 2 and confidence 0. For example, our tool reported a violation in release 0.2.0 involving `TaskTracker`, `TaskInProgress`, `JobTracker`, `JobInProgress`, and `MapOutputFile`. However, none of these classes were mentioned in the reconstructed refactorings. We searched Hadoop's MRs and found an open request *MAPREDUCE-278*, entitled "Proposal for redesign/refactoring of the `JobTracker` and `TaskTracker`". The MR states that the `TaskTracker`, `TaskInProgress`, `JobTracker`, and `JobInProgress` code is *hard to maintain*, *brittle*, and *merits some rework*. In addition, the MR mentions that the poor design of these components caused various defects in the system. Although our tool also reports `MapOutputFile` in the violation, it is not directly mentioned in the MR and may not actually be a part of this design violation.

3. Violations that are false positives: Violations in this category are not matched against any valid API refactorings or modifications requests, suggesting that such violations are not violations (false positives). However, not being developers of Hadoop, we cannot be certain whether such a violation is problematic or not. We found 184 violations belonging to this category using support of 2 and confidence of 0. As an example, in release 0.4.0, our tool reports a violation containing `ClientProtocol`, `NameNode`, `FSNamesystem`, and `DataNode`. Although there may actually be a design violation involving these components, we were unable to identify it. We believe the reason why these components are reported as a violation is because `ClientProtocol` contains a public field with the protocol version number and whenever the protocol changes, this number needs to change. Since `NameNode`, `DataNode`, and `FSNamesystem` implement the protocol, changes to them induce a change to `ClientProtocol`.

4. Violations that are false negatives: Some reconstructed refactorings are not matched to any violations identified by our tool. In such cases, we say that our tool failed to identify a refactoring opportunity (false negative). For example, the refactoring results show that in version 0.15.1, the `INode` inner class of `FSDirectory` was extracted into a separate class. `INode` provided the functionality for both directories and files. To improve this design, the designers moved the class out of `FSDirectory` and derived two sub-types `INodeFile` and `INodeDirectory`. This allowed the `DFSFileInfo` and `BlocksMap` classes to use specific subtypes of `INode`. The `INode` is a design rule to both `DFSFileInfo` and `BlocksMap`, and this was a design violation due to the lack of separation of concern in the `INode`. However, our approach did not identify a violation between these three classes because they were only involved in a single MR during the time frame we examined. The dearth of MRs involving these components suggests that these components do not belong to an actively-developed part of the system, and was not identified by our approach.

In Section 5, we further discuss the causes of false positive and false negative results.

Timing of Design Violation Detection.

For the set of confirmed violations in Hadoop, our approach identified a violation, on average, six releases prior to the release that the classes, in the reported violations, were actually refactored. Furthermore, there were also several violations that later appeared in open modification requests that describe needs for refactorings those violations, signifying the event that developers recognized the design problems. Figure 7 shows the distribution of the confirmed violations, by version, for Hadoop with minimum support of 2 and confidence of 0. Each point in the plot represents a set of confirmed violations, such that the horizontal axis shows the version that the violations were first identified by our approach and the vertical axis shows the version that the violations were refactored. Points above 20 in the vertical axis signify that the violations have not been refactored. For example, from the first release, our approach identified a number of violations, some of which were refactored in version 0.14.0 and some of which are not yet refactored.

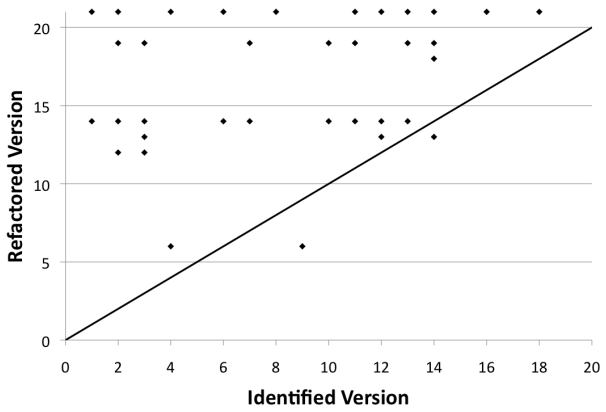

Figure 7: Timing of Violation Detection (Hadoop)

Since most of the points in Figure 7 are above the line, it indicates that our approach can identify design violations early in the development process. Our evaluation found, unsurprisingly, that increasing the support and confidence thresholds delayed the identification of violations and reduced time between our identification of a violation and its refactoring. We notice from Figure 7 that a small number of violations were not identified until after a refactoring occurred. A reason for this is that sometimes refactorings may not completely resolve a design violation among components. Although we associated the identified violations with past refactorings, this may indicate further refactoring may be required but has not been performed for us to detect it. For example, we detected a code clone issue in version 0.12.0 of Hadoop and these clones were refactored over several subsequent versions, indicating that the initial refactoring did not completely resolve the problem.

For Derby, our tool reported 4 of the confirmed violations in the sixth version that we studied. The current release of Derby is four versions afterward but the violations have not been refactored yet.

Characteristics of Identified Design Violations.

Most of the violations identified by our approach were caused by tight coupling or cyclic dependencies, although the refactoring techniques [10] to resolve them may differ. Four other types problems identified by our approach as violations are poor inheritance hierarchy, the need for push-down refactoring, the need for pull-up refactoring, and cloned code. Using a minimum support of 2 and confidence of 0, we identified 151 confirmed violations in Hadoop. Of these, 106 were due to tight coupling or cyclic dependencies, 27 from the need for pull-up refactoring, 31 from the need for push-down refactoring, and 59 from cloned code. Obviously from these numbers, many of the violations fit into more than one of these categories. With a minimum support of 2 and confidence of 0, we identified 21 confirmed violations in Derby. Of these 9 were due to tight coupling or cyclic dependencies and 12 were due to poor inheritance hierarchy.

As an example of a poor inheritance hierarchy identified by our approach, the `Statement` class in Derby has `PreparedStatement` and `CallableStatement` as subclasses. Our tool identified these three components in a violation. Rather than use polymorphism for the correct invocation of certain methods, two boolean fields are stored in the `Statement` class, `isPreparedStatement_` and `isCallableStatement_`, to denote the actual class type. To acknowledge this design violation, the developer wrote a comment next to one of the fields saying that "we can get rid of this member once we define polymorphic [methods]."

Consider an example of identifying the opportunity to applying a push-down method refactoring technique [10] in Hadoop. The `DFSClient`, `ClientProtocol`, and `NameNode` were in a violation in version 0.2.0 and matched with an API refactoring in version 0.14.0. In this refactoring, the `getHints` method was pushed down from the `ClientProtocol` to its subclass, `NameNode`. Since `DFSClient` was the only user of this method, it appeared that the method was not needed by other subclasses of `ClientProtocol` and was refactored.

An example of a violation for the need to apply a pull-up method refactoring, in Hadoop, involves the `DistributedFileSystem` and `FileSystem` classes. Our tool identified this violation in version 0.2.0. In version 0.12.0, several methods in `DistributedFileSystem` were pulled up to its parent, `FileSystem`. Since these methods were generic enough to apply to all file system types, they were made available to the other `FileSystem` subtypes.

A code clone example in Hadoop identified by our tool involved in the classes `Task`, `MapTask`, and `ReduceTask`. Our tool identified, in version 0.12.0, two violations: one involving `MapTask` and `Task`, and the other involving `ReduceTask` and `Task`. Various methods and inner classes from `ReduceTask` and `MapTask`, which were code clones, were pulled up to the parent `Task` class in versions 0.13.0, 0.14.0, and 0.18.0.

5. DISCUSSION

This section discusses our evaluation results, threats to validity, and limitations of our approach.

The quality of our violation identification depends heavily on the availability of modifications requests *and* their associated change sets. It is not surprising then, that we could not identify any violations from JEdit, given its mere 37 MRs. We found few violations in Derby due to the size of the system compared to the number of MRs. Although Derby is almost five times larger than Hadoop (in lines of source code), we found less than half the number of MRs to use as input. Given the large size of the system and the few MRs available, the likelihood of any two MR change sets containing the same files is very low and therefore the number of recurring patterns among the discrepancy sets, for deriving violations, is also low. In addition, our approach also relies on the quality of SCM historical data for computing change coupling; however, Derby was originally developed as a commercial product and the first eight years of its development history was unavailable for us to compute more accurate change coupling information.

Since we only applied our approach to three subject systems, we cannot conclude that the effectiveness of our design violation detection approach generalizes to all software systems; however, we did choose projects of different sizes and domains to begin addressing this issue. In addition, we cannot guarantee that the modification requests used in the evaluation are not biased. As Bird et al. [2] showed, the MRs that have associated change sets may not be representative of all the MRs in the system. For example, although we claim to identify design violations for actively-developed parts of a system, the collected MRs may not include the most active parts of the system and would diminish the effectiveness of our approach in reducing the maintenance costs incurred by design violations.

The accuracy of our approach also depends on how accurate the ACN model embodies design decisions and their assumption relations. For example, the ACN model we used in this paper were automatically generated from UML class diagrams derived from the source code. Some dependencies can only be reflected in other design models, such as an architectural description language. It is possible that these dependencies are missing from the ACN model, hence causing false positives. The violation we discussed in the previous section that contains `ClientProtocol`, `NameNode`, `FSNamesystem`, and `DataNode` is such an example. A future work is to evaluate the effectiveness of our approach using ACNs generated by combining high-level architectural models with source code.

We did not evaluate the recall of violations identified by our approach. In other words, we did not count the number of violations *not* identified by our approach. There are two reasons for this. First, it is not possible to manually identify *all* design violations in a non-trivial-sized software system, to determine the violations that our approach misses. Second, since the number of design violations is likely to be very large in non-trivial-sized software, it may be too expensive for an organization to resolve all the violations. While many existing approaches attempt to identify *all* refactoring opportunities, our approach uses modification requests and change coupling data to identify only violations in actively-developed parts of the system—the ones that are incurring maintenance costs. In the same vein, the violations identified by our approach can be prioritized for investigation based on support and confidence values because the higher the support for a violation, the more MRs led to its identification. Figure 5 corroborates this idea of prioritizing the violations, since a higher precision of confirmed violations were found with higher support and confidence values.

It is possible to use *design structure matrices* [1, 14], automatically derived from either source code or design models, to identify certain kinds of violations, such as cyclical

dependencies. However, we observe that some cyclical dependencies shown in DSMs are not problematic designs or they reside in stabilized, inactive part of the system, hence do not need to be refactored. Additionally, a large number of refactoring opportunities, such as code clones, cannot be reflected by DSM models.

6. CONCLUSION

Unintended violations of design assumptions may not be easily detectable by traditional software verification and validation techniques, but could cause modularity decay and expensive refactoring. To help software designers identify such violations early in the development process, we contributed an algorithm to predict the impact scope of a modification based on the modular structure framed by inferred design rules. We also contributed a framework that identifies problematic design violations by computing the discrepancies between design rule-based impact scope prediction and the change couplings revealed by how modification requests were actually fulfilled.

We evaluated our approach by investigating the version histories of Hadoop, Derby, and JEdit. Our evaluation shows that our approach can identify design violations accurately and the more modification requests are there, the more accurate our approach performs. It shows that problematic violations can be detected much earlier than manual identification by developers, showing the potential of preventing modularity decay early in the development process.

7. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants CCF-0916891 and DUE-0837665.

8. REFERENCES

[1] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.

[2] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced? bias in bug-fix datasets. In *Proc. 17th FSE*, pages 121–130, Aug. 2009.

[3] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society, 1996.

[4] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, Aug. 2006.

[5] Y. Cai and K. J. Sullivan. Simon: A tool for logical design space modeling and analysis. In *Proc. 20th ASE*, pages 329–332, Nov. 2005.

[6] Y. Cai and K. J. Sullivan. Modularity analysis of logical design models. In *Proc. 21st ASE*, pages 91–102, Sept. 2006.

[7] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *TSE*, 35(6):864–878, July 2009.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *TSE*, 20(6):476–493, June 1994.

[9] B. Fluri, H. C. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *Proc. 5th WICSA*, pages 66–74, Sept. 2005.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, July 1999.

[11] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. 14th ICSM*, pages 190–197, Nov. 1998.

[12] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Proc. 13th CSMR*, pages 255–258, Mar. 2009.

[13] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. In *Proc. 5th PROFES*, pages 220–233, Apr. 2004.

[14] S. Huynh, Y. Cai, Y. Song, and K. Sullivan. Automatic modularity conformance checking. In *Proc. 30th ICSE*, pages 411–420, May 2008.

[15] M. Kim. *Analyzing and Inferring the Structure of Code Changes*. PhD thesis, University of Washington, 2008.

[16] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Proc. 29th ICSE*, pages 333–343, May 2007.

[17] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. Duchien. A domain analysis to specify design defects and generate detection algorithms. In *Proc. 11th FASE*, pages 276–291, Mar. 2008.

[18] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *Proc. 5th MSR*, pages 1–5, May 2005.

[19] M. P. Robillard. Topology analysis of software dependencies. *TOSEM*, 17(4):18:1–18:36, Aug. 2008.

[20] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *TSE*, 17(2):141–152, Feb. 1991.

[21] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *Proc. 12th CSMR*, pages 329–331, Apr. 2008.

[22] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities. In *Proc. 13th CSMR*, pages 119–128, Mar. 2009.

[23] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *TSE*, 35(3):347–367, May 2009.

[24] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. 21st ASE*, pages 231–240, 2006.

[25] S. Wong and Y. Cai. Predicting change impact from logical models. In *Proc. 25th ICSM*, pages 467–470, Sept. 2009.

[26] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th ASE*, pages 197–208, Nov. 2009.

[27] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A hybrid approach to identify framework evolution. In *Proc. 32nd ICSE*, May 2010.

[28] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *TSE*, 30(9):574–586, Sept. 2004.

[29] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. 26th ICSE*, pages 563–572, May 2004.