

Detecting Software Modularity 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

This paper presents CLIO, an approach that detects *modularity violations*, which can cause software defects, modularity decay, or expensive refactorings. CLIO computes the discrepancies between how components *should* change together based on the modular structure, and how components *actually* change together as revealed in version histories. We evaluated CLIO using 15 releases of *Hadoop Common* and 10 releases of *Eclipse JDT*. The results show that hundreds of violations identified using CLIO were indeed recognized as design problems or refactored by the developers in later versions. The identified violations cover multiple symptoms of poor design, some of which are not easily detectable using existing approaches.

Categories and Subject Descriptors

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

Keywords

modularity violation detection, refactoring, bad code smells, design structure matrix

1. INTRODUCTION

The essence of software modularity is to allow for independent module evolution and independent task assignment [1, 18]. In reality, however, two modules that are supposed to be independent may always change together, due to unwanted side effects caused by quick and dirty implementation. 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 [15]. Such activities cause modularity decay over time and may require expensive system-wide refactorings. Though empirical studies have revealed a strong correlation

between software defects and eroding design structure [5, 22], traditional verification and validation techniques do not find modularity violations because these violations do not always influence the functionality of software systems directly.

This paper presents CLIO, an approach that detects and locates *modularity violations*. CLIO compares how components *should* change together based on the modular structure and how components *actually* change together as reflected in the revision history. The rationale is that, if two components always change together to accommodate modification requests,¹ but they belong to two separate modules that are supposed to evolve independently, we consider this as a *modularity violation*.

CLIO has three parts. First, we leverage Baldwin and Clark’s design rule theory and design structure matrix (DSM) [1] to manifest independently evolvable modules, from which we determine *structural coupling*—how components should change together. Second, we mine the project’s revision history to model *change coupling*—how components actually change together [9]. We identify modularity violations by comparing the results of *structural coupling* based impact scope analysis with the results of *change coupling* based impact scope analysis.

We applied CLIO to the version histories of two large-scale open source software systems: 15 releases of *Hadoop Common*,² and 10 releases of *Eclipse JDT*.³ Our evaluation strategy was to identify *violations* for each pair of releases. If a violation was indeed problematic, it is possible that developers recognized and fixed it in a later release through a refactoring. We considered a detected violation as being *confirmed* if it was indeed addressed or recognized by developers later. We used two complementary evaluation methods. First, we compared the detected violations with refactorings automatically reconstructed using Kim et al.’s API matching technique [16]. Second, for the remaining detected violations, we manually examined modification requests to see whether those violations were at least recognized by developers. Because it is possible that the violations detected in recent versions are not recognized by the developers yet, we also manually examined the corresponding code to determine whether the code shows any symptoms of poor design.

We identified 231 violations (47%) from the 490 modification requests of *Hadoop*, of which 152 (65%) violations were confirmed. From the 3458 modification request of *Eclipse*

¹Consistent with Ying et al. [30], a modification request can be a bug fix or feature enhancement.

²<http://hadoop.apache.org/common/>

³<http://www.eclipse.org/jdt/>

JDT, CLIO identified 399 violations (12%), which shows that the changes in Eclipse match its modular structure better. Among these violations, 161 (40%) were confirmed. The results also show that CLIO identifies modularity violations much earlier than manual identification by developers so that designers can be alarmed to avoid accumulating modularity decay. Third, the identified violations include symptoms of poor design, some of which cannot be easily detected using existing approaches.

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

2. RELATED WORK

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

Automatic Detection of Bad Code Smells.

Fowler [8] describes the concept of *bad smell* as a heuristic for identifying redesign and refactoring opportunities. Example bad smells include code clone and feature envy. Garcia et al. [11] 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. Several other techniques [24–26] automatically identify bad smells that indicate needs of refactorings. For example, Tsantalis and Chatzigeorgiou’s technique [25] 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.

CLIO’s modularity violation detection approach is different in several aspects. First, it is not confined to particular types of bad smells. Instead, we hypothesize that multiple types of bad smells are instances of modularity violations that can be uniformly detected by CLIO. For example, when code clones change frequently together, CLIO will detect this problem because the co-change pattern deviate from the designed modular structure. Second, by taking version histories as input, CLIO detects violations that happened most recently and frequently, instead of bad smells detected in a single version without regard to the program’s evolution context. Similar to CLIO, Ratzinger et al. [19] also detect bad smells by examining change couplings. Their approach leaves it to developers to identify design violations from visualization of change coupling, while CLIO locates violations by comparing change coupling with structural coupling. The detected violations thus either reflect the problem in the original design or introduced in the subsequent modification requests.

Design Structure Matrix Analysis.

The most widely used *design structure matrix* (DSM) tools

include Lattix,⁴ Struture 101,⁵ and NDepend.⁶ These tools support automatic derivation of DSMs from source code in which columns and rows model classes or files, and the dependencies model function calls, inheritance, etc. Different from these tools, the DSMs used in CLIO are generated from *augmented constraint networks* (ACNs) [3,4], which separate the interface and implementation of a class into two design dimensions, and manifest implicit and indirect dependencies [27,28]. Our prior work shows that an ACN-derived DSM can capture more types dependencies than that of a syntactical DSM. The detail description on ACN is described elsewhere [3].

Sangal et al. [21] identify modularity violations using Lattix DSMs. Using Lattix, the user can specify which classes should not depend on, that is, syntactically refer to, which other classes. The tool raises an alarm if such predefined constraints are violated. A key difference between CLIO and Lattix violation detection techniques is that CLIO uses version histories as opposed to analyzing a single version only. CLIO detects violations that occur during software evolution, many of which are not in the form of syntactical dependency, and thus will not be detected by Lattix. Another major difference is that CLIO takes recency and frequency into consideration when identifying modularity violations.

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 [22], Cataldo et al. [5]) have found that modules with lower coupling are less likely to contain defects than those with higher coupling. Various metrics have been proposed (e.g., Chidamber and Kemerer [6]) to measure coupling and failure proneness of components. The relation between change coupling [9] and defects has also been recently studied. Cataldo et al.’s [5] study revealed a strong correlation between density of change coupling and failure proneness. Fluri et al.’s [7] study shows that a large number of change coupling relationships are not entailed by structural dependencies. While the purpose of these studies are to statistically account for the relationship between software defects, change couplings, and syntactic dependencies, CLIO’s purpose is to *locate* modularity violations that may cause design decay and software defects.

3. MODULARITY VIOLATION DETECTION

Consider a project that evolves from version n to $n+1$. A number of modification requests (MRs) are fulfilled during this period, including both bug fixes and feature enhancements. Suppose that, before the release $n+1$ is publicly released, the project manager needs to make sure that the modular structure of the system is well-maintained, that is, unexpected dependencies were not introduced by quick and dirty maintenance. Fixing these problems would prevent modularity decay. We now introduce how our modularity violation detection approach, supported by the CLIO framework,⁷ can help achieve this goal.

⁴<http://www.lattix.com/>

⁵<http://www.headwaysoftware.com/products/structure101/>

⁶<http://www.ndepend.com/>

⁷CLIO is the Greek muse of history.

3.1 Framework Overview

Figure 1 depicts an overview of CLIO that takes the following artifacts as input. The first input is the original modular structure of version n before implementing these modification requests. Since an accurate design model is usually not available in practice, CLIO uses the *Moka* [27] tool to reverse-engineer UML class diagrams from compiled Java binaries. CLIO then uses the *uml2acn* [27,28] tool to transform a class diagram into an *augmented constraint network* (ACN) [3], a design model that formalizes the key concepts of Baldwin and Clarke’s design rule theory [1], which we introduce soon. From an ACN, a *design structure matrix* (DSM) can be automatically derived [3,4].

The second input is the revision history of the project, which is used to derive *change couplings* from a set of files changed to implement modification requests. The *extract* plugin of CLIO computes change couplings at a file level, following the technique of Ying et al. [30].

The third input is the detailed information about a set of files S (called the MR solution), which was modified to fulfill each modification request.

For each modification request, CLIO’s *dr-predict* plug-in outputs the components that *are likely* to be changed according to the original modular structure (FileSet A in Figure 1). CLIO’s *logic-predict* plugin also reports the components that are likely to be changed according to co-change patterns, recorded in FileSet B. Finally, given A and B , and a MR solution S , the *detect* plug-in computes a set of discrepancies, $D = (B \cap S) \setminus A$. By using $B \cap S$, CLIO filters out files that were accidentally changed together. Recurring discrepancies (a subset of files in D) are then reported to the users as *violations*.

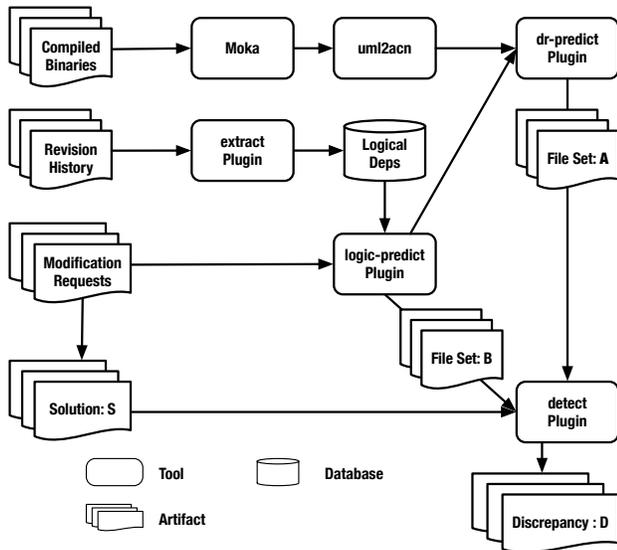


Figure 1: Approach Overview: the CLIO Framework

3.2 History-based Impact Scope Analysis

CLIO takes source code revision history as input and extracts *change couplings* between files, storing the support and confidence values between files into a database follow-

ing Ying et al. [30] and Zimmermann et al. [31].⁸ It reads the change couplings from the database and predicts the impact scope (noted as FileSet B in Figure 1) from the starting change set. 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 th_s and confidence th_c thresholds.

For each modification request m , CLIO first selects a subset of files in the corresponding change set that exhibit the strongest co-change patterns according to the change coupling analysis. We call this selected set of files, the starting change set, σ , as the discrepancy between σ ’s impact scope based on structural couplings and σ ’s impact scope based on change couplings is mostly likely to reveal modularity violations.

3.3 Background

This section illustrates key background concepts of CLIO’s modularity-based impact scope analysis using a maze-game example described by Gamma et al [10]. Figure 2 depicts a UML class diagram for a maze game example used in our prior work [28]. A maze consists of a set of rooms that know their neighbors, a wall or a door to another room. The base class, *MapSite*, captures the commonality of all the maze components. The diagram shows the abstract factory pattern to support two variations of the game: an enchanted maze game and a bombed maze game.

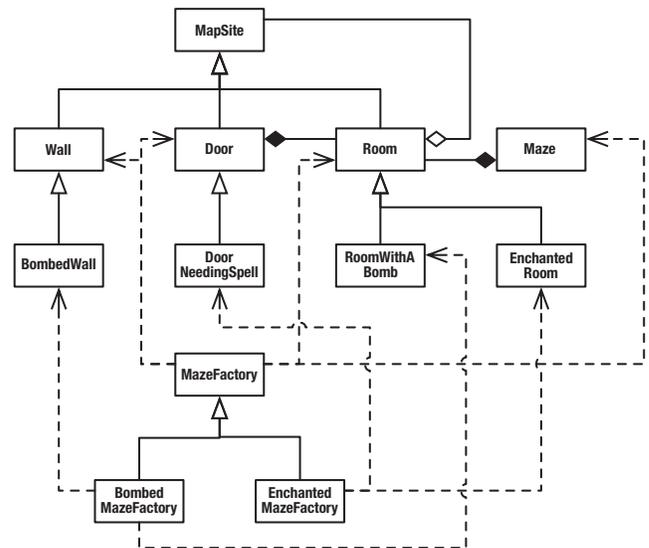


Figure 2: Maze game UML class diagram [28]

⁸A transaction is defined as an atomic commit in a version control repository (e.g. Subversion). For repositories that do not natively support the concept of transactions (e.g. CVS), heuristics and techniques have been developed to reconstruct transactions. Consistent with Zimmermann et al. [31], the frequency of a set in a set of transactions T is $frq(T, x) = |\{t \in T, x \subseteq t\}|$. 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 $conf(T, x_1 \Rightarrow x_2) = \frac{frq(T, x_1 \cup x_2)}{frq(T, x_1)}$.

Augmented Constraint Network (ACN).

An ACN [3,4] makes design decisions and their relations, which are implied in a UML class diagram, explicit. Figure 3 shows part of an ACN derived from the above UML class diagram. Each class is modeled using two variables (lines 1–6): an interface variable⁹ ending with `_interface` and an implementation variable ending with `_impl`. Each variable has a two-value domain modeling a current decision and an unknown possibility. Lines 7 to 9 show several sample assumption relations. For example, since `Room` inherits from `MapSite`, its implementation makes assumption on both the interface and implementation of `MapSite` (lines 7, 8).

Dominance relations in an ACN describe asymmetric dependency relationships among design decisions, the essence of Baldwin and Clark’s concept of *design rules* [1]. Baldwin and Clark coined a term, *design rules*, to refer to stable design decisions that decouple otherwise coupled design decisions, hiding the details of subordinate components. We emphasize that Baldwin and Clark’s concept of *design rule* is different from the concept of *rules* used in other areas, such as the rules of not creating clones or cyclic dependencies, but rather they are essentially generalized interfaces between components. Example design rules include abstract interfaces, application programming interfaces (APIs), or a shared data format agreed among development teams [23]. Broadly speaking, all non-private parts of a class used by other classes can be seen as design rules.

For example, line 11 models that `Room`’s implementation decision cannot influence its interface design, which is a *design rule*. One should not arbitrarily change `Room`’s interface to improve its implementation because other components may depend on it. In our previous work, we defined eight heuristics to automatically derive dominance relations from reverse-engineered UML diagrams. Here, we mention one concrete heuristic which is based on inheritance. Dependencies of a UML class diagram, such as method calls and object aggregations, are used to derive constraints in the ACN. The details on all the heuristics is described in our previous paper [14].

- | | |
|-----|---|
| 1. | <code>MapSite_interface</code> : { <i>orig</i> , <i>other</i> } |
| 2. | <code>MapSite_impl</code> : { <i>orig</i> , <i>other</i> } |
| 3. | <code>Room_interface</code> : { <i>orig</i> , <i>other</i> } |
| 4. | <code>Room_impl</code> : { <i>orig</i> , <i>other</i> } |
| 5. | <code>Maze_interface</code> : { <i>orig</i> , <i>other</i> } |
| 6. | <code>Maze_impl</code> : { <i>orig</i> , <i>other</i> } |
| 7. | <code>Room_impl</code> = <i>orig</i> \Rightarrow <code>MapSite_interface</code> = <i>orig</i> |
| 8. | <code>Room_impl</code> = <i>orig</i> \Rightarrow <code>MapSite_impl</code> = <i>orig</i> |
| 9. | <code>Maze_impl</code> = <i>orig</i> \Rightarrow <code>Room_interface</code> = <i>orig</i> |
| 10. | (<code>MapSite_impl</code> , <code>MapSite_interface</code>) |
| 11. | (<code>Room_impl</code> , <code>MapSite_interface</code>) |
| 12. | (<code>Maze_impl</code> , <code>Room_interface</code>) |

Figure 3: Maze game augmented constraint network [28]. Only a part of ACN is shown for presentation purposes.

⁹An *interface* variable in an ACN represents the publicly accessible methods, fields, etc. of a class. It should not be confused with the *programmatically interface* construct provided by many object-oriented languages.

Design Structure Matrix (DSM).

Figure 4 shows a sample design structure matrix (DSM) automatically derived from maze game ACN. A DSM is a square matrix whose columns and rows can be labeled with design variables of an ACN. Each cell marked with “x” represents a pair-wise dependency relation: if y depends on x , the cell on row y , column x will be marked. For example, cell (r11, c2) indicates that `Room_impl` depends `MapSite_interface`. Cell (r2, c11) is not marked because `MapSite_interface` dominates `Room_impl` as a design rule.

Design Rule Hierarchy (DRH).

In order to identify modules— independent task assignments according to Parnas’ definition [18], our prior work created a special clustering based on ACN called the *design rule hierarchy* (DRH). Using this clustering, the columns and rows of the DSM can be reordered into *layers*, that is, a lower triangle form in which the top right corner is blank. The first layer in a DSM, l_1 , is the group of variables clustered at the top left corner, and does not depend on any other layers. A layer l_n only depends on layers l_{n-1} to l_1 . In a DRH, each layer contains a set of *modules* that are independent from each other. In the DSM, the *modules* are inner groups of variables along the diagonal, and there are no dependencies between the modules within the same layer.

For example, Figure 4 shows a DSM after the clustering process and each identified layer is denoted as an outer rectangle in bold line along the diagonal line. This DRH has four layers in total: The first layer (r1-2, c1-2) contains the most influential design rules that must remain stable. In other words, changing the top-level design rules, `Maze_interface` and `MapSite_interface`, can have drastic effects on the system. The second layer (r3-6, c3-6) contains decisions that only depend on the top layer decisions (r1-2, c1-2). Similarly, the third layer (r7-13, c7-13) contains decisions that make assumptions about the decisions within the first two layers only.

Within each layer, there are inner rectangles along the diagonal line such as (r1, c1) or (r7-8, c7-8). They are *modules* containing decisions that can be made in parallel because there are no inter-module dependencies within a layer. For example, `MazeFactory_interface` (r7) and `MazeFactory_impl` (r8) decisions can be made in parallel with other inner decisions of the same layer, such as `DoorNeedingSpell_interface` (r12). The modules in the last layer (r14-24, c14-24) can be designed concurrently with each other, and can be swapped out for different implementations without affecting the rest of the system. For example, the task of designing an enchanted maze game (r16-17) and the task of designing a bombed maze game (r20-21) can be independently accomplished.

These hierarchical relationships among design decisions captured in a design rule hierarchy can also be represented as a directed acyclic graph where each vertex u corresponds a module in the DSM, containing a set of decisions, and each edge ($u \rightarrow v$) defines that changing a module u may affect a module v . Figure 5 shows a part of the design rule hierarchy graph derived from the maze game ACN.

3.4 Modularity-based Impact Scope Analysis

Taking a DRH graph and starting change set σ as input, CLIO analyzes σ ’s impact scope to identify the components that should change together according to the modular struc-

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Maze_interface	1	.	.																					
MapSite_interface	2		.																					
Wall_interface	3		x	.																				
Door_interface	4		x		.																			
Room_interface	5		x			.																		
MapSite_impl	6		x				.																	
MazeFactory_interface	7						.																	
MazeFactory_impl	8	x	x	x	x	x		x	.															
BombedWall_interface	9		x	x						.														
EnchantedRoom_interface	10		x			x					.													
Room_impl	11		x			x	x					.												
DoorNeedingSpell_interface	12		x		x								.											
RoomWithABomb_interface	13		x			x								.										
BombedWall_impl	14		x	x			x			x					.	x								
Wall_impl	15		x	x			x								.									
EnchantedRoom_impl	16		x			x	x				x	x				.								
EnchantedMazeFactory_impl	17	x	x	x	x	x		x	x		x	x					.	x						
EnchantedMazeFactory_interface	18							x										.						
RoomWithABomb_impl	19		x			x	x				x	x							.					
BombedMazeFactory_interface	20							x												.				
BombedMazeFactory_impl	21	x	x	x	x	x		x	x	x										x	.			
Maze_impl	22	x	x			x															.			
DoorNeedingSpell_impl	23		x		x	x	x					x										.	x	
Door_impl	24		x		x	x	x															.	.	

Figure 4: Maze game DSM [28]

ture if σ changed. CLIO uses Robillard’s [20] relevant artifact recommendation algorithm that identifies a subset of nodes in a graph, relevant to an initial set of interests based on the graph’s topology. We chose this algorithm because the input format and the algorithm requirements are similar: (1) a DRH is an acyclic graph just like a static dependency graph in Robillard’s and (2) the algorithm must carefully propagate the degree of relevance (weights) along edges until they are stabilized using an iterative, fix-point algorithm.

In order to demonstrate our modularity-based change scope analysis approach, we depict a small subset of the maze game DRH graph in Figure 5 for the purpose of illustration. In Figure 5, we only show 1 of the 2 modules in layer 1, 3 modules each from layer 2 and 3, and 1 module from layer 4. Note that the edges of the DRH graph are populated based on constraints in the ACN as introduced in our prior work [28].

The vertices with shaded background and white text model the starting change set within a modification request (MR). Beginning with the starting change set, we assign a weight μ , in the range $[0, 1]$, to each vertex, in a breadth-first order. The starting change set vertices are assigned the maximum weight of 1 and added to a initial set of interests, S . From vertex `Room_interface`, we examine its neighbors, the subordinating decisions that `Room_interface` influences, and assign them a weight. While traversing the graph to assign weights, we ignore the starting change set’s design rules to ensure that they remain stable. For example, since the `Room` class is the starting change set (row 5 and row 11 in the DSM) in our example, then its design rules, `MapSite`’s interface and implementation should not be within their impact scope.

Robillard [20] defines a formula for computing the weight of a vertex:

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

Using this formula, we assign higher weights to vertices that

share more edges with elements in the set of interest S . This allows us to identify the components that are likely to be affected by the starting change set due to the strengths of their design-level dependencies. μ is a weight and α is a constant defined to determine the degree of relevancy propagation (0.25 in our evaluation).

To start the each iteration of the 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. 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 5 shows the weights for each vertex after all weights are propagated. The vertices whose weights are above the threshold th_d (e.g., 0.75) are then recommended as being in the impact scope (noted as node with dotted circles). Below, we discuss how this minimum threshold th_d is determined.

3.5 Discrepancy Analysis

We vary the thresholds to find values that maximize measure of accuracy over all the MRs. With *dr-predict*, we vary the minimum weight threshold th_d from 0 to 0.95 in increment of .05. With *logic-predict*, we vary th_s from 2 to 10 and th_c from 0 to 0.95 in increment of .05.

For each MR, we compute discrepancies between structural coupling based impact scope and change coupling based impact scope. We then identify recurring discrepancies over several versions of the software by using a frequent-pattern mining algorithm [12]. The recurring patterns among these discrepancies are called *modularity violations*. Consider two MRs with the same starting change set of a . Suppose that the set of discrepancies is $\{\{a,b,c\}, \{a,b\}\}$. Then, we say that $\{a,b\}$ is a modularity violation that occurred twice, and $\{a,b,c\}$ is a modularity violation that occurred once.

For example, `EnchantedMazeFactory_impl` and `BombedMazeFactory_impl` are both located in the last layer of the DSM, meaning that they should evolve independently from

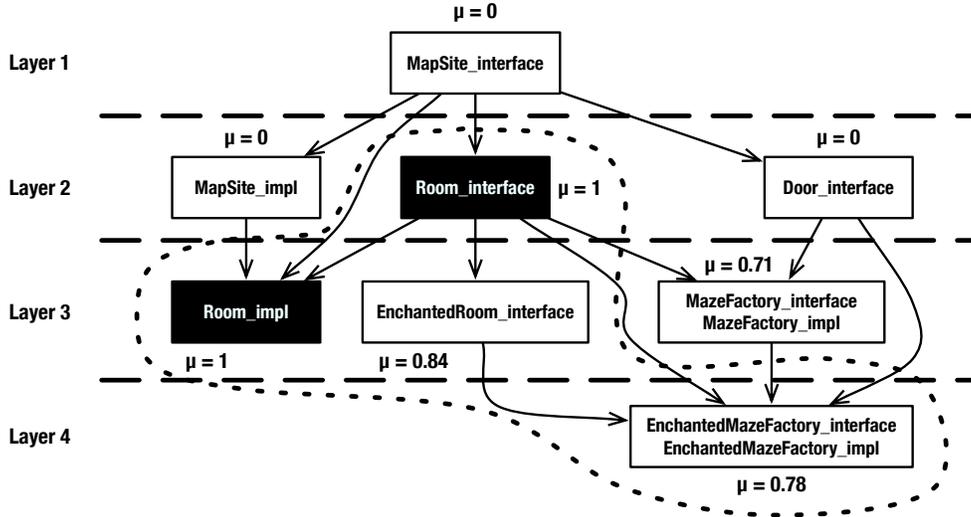


Figure 5: Maze game design rule hierarchy

each other. CLIO’s *dr-predict* plugin will never report that they are in each other’s change scope. If the revision history shows that they always change together, e.g. due to similar changes to cloned code, CLIO will report that there is a *modularity violation*. Consider another example, since `MapSite_interface` is the design rule of `Room_impl`, it is normal that `MapSite_interface` changes and influences `Room_impl` along with other dependent components. But CLIO’s *dr-predict* plugin will never predict `MapSite_interface` to be within the change scope of `Room_impl`. However, if the revision history shows that whenever `Room` changes, `MapSite` always changes with it, it is a violation because all other components that depend on `MapSite` may be affected, causing unwanted side effects.

4. EVALUATION

To assess the effectiveness of CLIO’s modularity violation detection approach, the evaluation aims to answer the following questions:

Q1. How accurate are the modularity violations identified by CLIO? That is, do these identified violations indeed indicate problems? Given the difficulty of finding the designers of the subject systems who can most accurately answer this question, we evaluate CLIO retrospectively and conservatively: we examine the project’s version history to see whether and how many violations we identified in earlier versions are *indeed* refactored in later versions or recognized as design problems by the developers (e.g., through modification requests, source code comments). The *precision* calculated this way is the most conservative, lower-bound estimation because it is possible that some violations we identified have not been recognized by the developers yet, 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 CLIO identify problematic violations? Our purpose is to see if this approach can detect design problems early in the development process. Although it may not be necessary to fix a violation as soon as it appears,

revealing these violations will make the designers alarmed as soon as possible to avoid accumulating modularity decay. For each confirmed violation, we compare the version where it was identified with where it was actually refactored or recognized by the developers.

Q3. What are the characteristics of design violations identified by our approach? We also examined the detected violations’ corresponding code to see whether they show any symptoms of poor design and categorized the violations into four categories.

4.1 Subjects

We choose two large-scale open source projects, Hadoop Common and Eclipse Java Development Tools (JDT) as our evaluation subjects. Hadoop is a Java-based distributed computing system. We applied our approach to the first 15 releases, 0.1.0 to 0.15.0, covering about three years of development. Eclipse JDT is a core AST analysis tool kit in the Eclipse IDE. We studied 10 releases of Eclipse JDT, from release 2.0 to 3.0.2, also covering about three years of development. Our evaluation use both their revision histories and source code. For Hadoop, we investigated their SVN repository to extract transactions. Eclipse JDT used CVS instead of SVN, so we use the *cvs2svn*¹⁰ tool to derive the transactions. In Table 1, we present some basic data regarding to Hadoop and Eclipse JDT that we studied. We removed commits with only one file or more than 30 files because they either do not contribute to CLIO’s modularity violation detection or they include noise such as changes to license information. For each release pair n and $n+1$, we computed discrepancies between the results of structural-coupling based impact scope analysis and the results of change-coupling based impact scope analysis. We then accumulated the discrepancies over the five most recent releases to identify recurring violations that occur more than a certain number of times. The experiments showed that the results do not matter if we aggregate discrepancies over more than 5 releases.

¹⁰<http://cvs2svn.tigris.org/>

Table 1: Characteristics of subject programs

Subjects	SLOC	#Transactions	#Releases	#MRs
Eclipse JDT	137K-222K	27806	10	3458
Hadoop	13K-64K	3001	15	490

Table 2: Modularity violations that occurred at least twice in the last five releases

	$ V $	$ V \cap R $	$ V \cap M $	$ CV $	Pr.
Eclipse JDT	399	55	104	161	40%
Hadoop	231	81	71	152	66%

4.2 Evaluation Procedure

We ran our experiments on a Linux server with two quad-core 1.6Ghz Intel Xeon processors and 8GB of RAM. We evaluate the output of CLIO, that is, a set of *violations*, by checking the source code and MR records in later versions to see if they were indeed refactored or recognized as having a design problem. If so, we call such violation as being *confirmed*. We use both automated method and manual inspection to confirm a violation.

First, we compared the detected 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. This tool 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. We chose this technique because it has a 5.01% higher precision than other similar techniques according our recent comparative study [29].

As these automatically reconstructed refactorings are method-header level refactorings, we aggregated them up to a class level to compare with the violations CLIO identified. We consider a violation as *confirmed* if it overlaps with any class-level refactorings. 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.

Second, 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, these reported violations through redesign or refactoring activities. For the rest of the reported violations, we studied the corresponding source code to see whether they include any symptoms of poor design.

4.3 Results

We analyzed our results by answering the questions proposed at the beginning of the section.

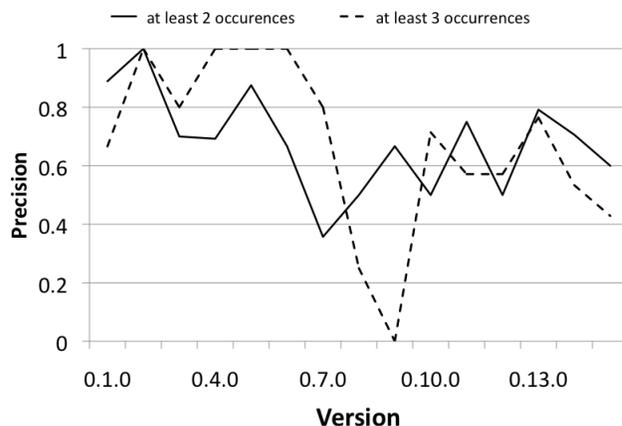
4.3.1 Q1. Accuracy of Identified Design Violations

Table 2 shows the total number of violations reported by CLIO ($|V|$), the total number of violations that match with automatically reconstructed refactorings ($|V \cap R|$), the total number of remaining violations that were confirmed based

on manual inspection ($|V \cap M|$), the total number of *confirmed violations* $|CV|$ (which is $|V \cap R| + |V \cap M|$), and the *precision*, which is defined as the number of confirmed violations out of the total number of reported violations: $\frac{|CV|}{|V|}$.

CLIO reported 231 violations that occur at least twice in a five release period in Hadoop, out of which 152 (66%) were confirmed. 81 of them were automatically confirmed and 71 were manually confirmed. Figure 6 shows the precision for those violations that occur at least twice and the violations that occur at least three times. With at least three occurrences, we obtain a similar precision of 67% but fewer reported violations. For Eclipse JDT, CLIO reported 399 violations, of which 161 were conservatively confirmed (40% precision). Requiring violations to occur at least three times increased the precision to 42%. We only discuss the results of requiring at least two occurrences for the rest of the paper because the results of higher occurrence rates are its subsets.

By comparing the results of Hadoop and Eclipse JDT, we first observe that Eclipse is better modularized and more stable: although Eclipse JDT is about 10 times larger than Hadoop, less than three times more refactorings were discovered from Eclipse JDT than from Hadoop, showing that it has been less volatile. This is consistent with the fact that only 12% of all the 3767 Eclipse MRs were detected to have violations (in Hadoop, the number is 47% out of the 490 MRs), showing that the changes to Eclipse JDT matches its modular structure better. Because Eclipse JDT is much larger and the violations found are much sparser, it was much harder for us to determine if a violation indicates a problem, hence leading to a lower precision.

**Figure 6: Precision (Hadoop)**

In-depth Case Study: Hadoop.

Now we present an in-depth study of Hadoop to demonstrate examples of violations that are (1) automatically confirmed violations, (2) manually confirmed violations, (3) false positives (violations that are not confirmed), and (4) false negatives (refactorings that are not identified as violations).

Automatically confirmed violations: In release 0.3.0, CLIO identified a violation involving `FSDirectory` and `FSNamesystem`. `FSNamesystem` depends on `FSDirectory.isValidBlock` method, but it often changes with `FSNamesystem`. 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, CLIO identified this violation 11 releases prior to the actual refactoring.

Manually confirmed violations: CLIO reported a violation in release 0.2.0 involving `TaskTracker`, `TaskInProgress`, `JobTracker`, `JobInProgress`, and `MapOutputFile` that does not match with automatically 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 these classes are “*hard to maintain, brittle, and merits some rework.*” The MR also mentions that the poor design of these components have caused various defects in the system.

False positive violations: Violations in this category cannot be confirmed either automatically or manually. In most cases, we cannot determine if there is a problem because we are not domain experts. As an example, in release 0.4.0, CLIO reports a violation containing `ClientProtocol`, `NameNode`, `FSNamesystem`, and `DataNode`. `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`. Although there may actually be a design problem, we are not able to determine it for sure.

False negative violations: Some reconstructed refactorings are not matched to any violations identified by CLIO. There are many micro refactorings that happen within a class and do not influence the macro structure of the system. Refactorings can also happen for other purposes.

Another reason is that some discrepancies only occur once, so CLIO cannot tell if they are accidentally changed together or there is a problem, but the developers may have realized and fixed it before it happens again. For example, in version 0.15.1, the `INode` inner class of `FSDirectory` was refactored and extracted into a separate class, and two of its sub-types `INodeFile` and `INodeDirectory` were created so that the `DFSFileInfo` and `BlocksMap` classes can be separated and use specific `INode` subtypes. CLIO did not identify a violation between these classes because they were only involved in a single MR during the time frame we examined.

4.3.2 Q2. Timing of Violation Detection

In Hadoop and Eclipse JDT, CLIO identifies a violation, on average, 6 and 5 releases respectively, prior to the releases where the classes involved in the violation were actually refactored. Figure 7 shows the distribution of the confirmed violations over Hadoop releases. 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 CLIO and the vertical axis shows the version that the violations were refactored or recognized by the developers. Points above 20 in the vertical axis signify that the violations have been recognized by developers but not refactored yet. Most of the points in Figure 7 are above the line, indicating that CLIO can identify design violations early in the development process so that the designers can be alarmed to avoid these problems accumulating into severe decay.

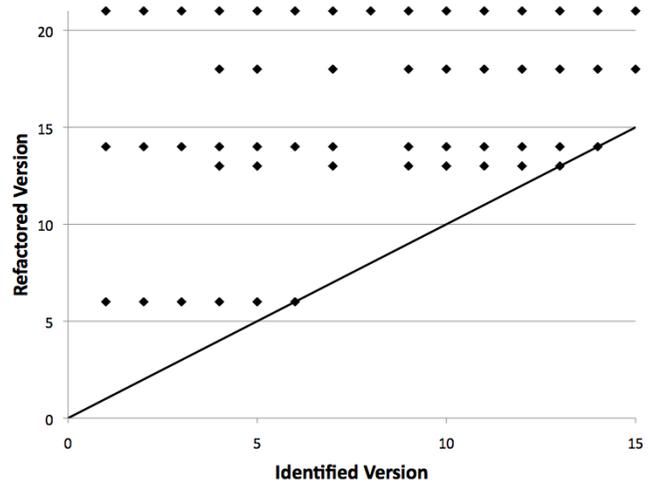


Figure 7: Timing of Violation Detection (Hadoop)

Table 3: Characteristics of the Violations

Subjects	cyclic	clone	inheritance	coupling
Eclipse JDT	72	52	19	25
Hadoop	58	18	37	66

4.3.3 Q3. Characteristics of Identified Violations.

We further analyzed the symptoms of design problems associated with the detected violations and categorized them into the following four types: (1) cyclic dependency, (2) code clone, (3) poor inheritance hierarchy, and (4) unnamed coupling. The first three symptoms are both well defined and can be detected using existing tools. We call the fourth category *unnamed* because they are not easily detectable using existing techniques, to the best of our knowledge. Table 3 shows the number of confirmed violations under each category in Hadoop and Eclipse JDT. The cyclic dependency, code clone, and unnamed coupling violations reported in the table are mutually exclusive from each other. The symptoms of poor inheritance hierarchy often overlap with cyclic dependency or unnamed coupling. Next we provide examples from each category.

Cyclic Dependency. Both systems contain considerable number of cyclic dependencies. For example, in Eclipse JDT, we found that the `JavaBuilder` and `AbstractImageBuilder` often change together, and the code shows that `JavaBuilder` contains a subclass of `AbstractImageBuilder`, and `AbstractImageBuilder` contains a `JavaBuilder`. In a Lattix DSM, there are no symmetric marks to alarm the designer of this indirect cyclical dependency. Similarly, we found that all of the following five files, or their subsets often change together: `JavaProject`, `DeltaProcessor`, `JavaModelManager`, `JavaModel`, and `JavaCore`. It turns out that these five classes form a strongly connected components if represented as a syntactic dependency graph.

Code Clone. Some modularity violations detected by CLIO involve code clones. In Hadoop version 0.12.0, the detected modularity violation involves the classes `Task`, `MapTask`, and `ReduceTask`. CLIO reported two violations: one involving `MapTask` and `Task`, and the other involving `ReduceTask` and `Task`. Various methods and inner classes from `ReduceTask` and `MapTask` were pulled up to the parent `Task` class in versions 0.13.0, 0.14.0, and 0.18.0. In Eclipse JDT

and Hadoop, there are 52 and 18 violations respectively that exhibit symptoms of code cloning. Using clone detectors, it is possible that a much larger number of code clones can be detected, but it may be too costly and not necessary to refactor all of them. CLIO picks up the ones that happen most recently and most frequently, and provides more targeted candidates to be refactored.

Poor Inheritance Hierarchy. The poor hierarchy violations we identified all have the symptoms that the subclasses causing the base class and/or other subclasses to change for different reasons. For example, we identified, in version 0.2.0 of Hadoop, a violation involving the `DistributedFileSystem` and `FileSystem` classes, which was refactored in version 0.12.0: several methods in `DistributedFileSystem` were pulled up to its parent, `FileSystem`, making them available to the other `FileSystem` subtypes. Another reason is that the subclasses extensively use some methods in their parent class and a push-down method refactoring should have been applied [8]. For example, in Hadoop version 0.14.0, the `getHints` method was pushed down from the `ClientProtocol` to its subclass, `DFSClient`, because it was the only user of this method. They were detected as a violation in version 0.2.0.

In some cases, the parent classes depend on the subclasses and form a cyclic dependency. In Hadoop version 0.1, modification request #51 describes changing the `DistributedFileSystem` class but its parent class `FileSystem` and another child of the `FileSystem`, `LocalFileSystem`, are also part of its solution. There are no syntactic dependencies between the two sibling classes. By release 0.3, CLIO reported that this modularity violation was observed more than three times already. The code shows that the parent `FileSystem` class contains methods to construct both of the two subclasses. The parent class is thus very unstable because changes to a child require changes to itself and its other children. Our intuition that this is a problematic issue was confirmed when we looked forward through the revision history and found that by release 0.19, the method to construct `DistributedFileSystem` had been deprecated in `FileSystem`, in favor of a method in an external class. As a similar example in Eclipse, `Scope` is the parent of `ClassScope` and `BlockScope`, but it constructs both of its children. We categorized this type of violation as both poor inheritance and cyclic dependency.

Unnamed Coupling. The files involved in violations of this category often change together, but they either do not explicitly depend on each other (and are not code clones), or have asymmetric dependency. For example, in Hadoop, `DatanodeInfo` and `DataNodeReport` were involved in a violation, and was later refactored. In the modification request comments, the developer says that these classes *seem to be similar* and need to be refactored.

The `FSDirectory` and `FSNamesystem` we mentioned earlier is also an example of unnamed coupling. CLIO detected this violation because the only allowed change order is from the interface of `FSDirectory` to `FSNamesystem`. But the revision history shows that changes to `FSNamesystem` often cause `FSDirectory` to change. In the corresponding syntactical DSM, these two classes reside in the same package, and `FSNamesystem` depends on `FSDirectory`. Using a Lattix DSM, the user can mark that `FSDirectory` should not depend on `FSNamesystem` so that if `FSDirectory` explicitly refers to `FSNamesystem`, Lattix will raise an alarm. How-

ever, in reality, `FSDirectory` never explicitly refers to `FSNamesystem`, although it often changes with `FSNamesystem`. Table 3 shows that in Hadoop 66 out of 152 of the confirmed violations fall into this category (In Eclipse, the number is 25 out of 161). We are not aware of existing techniques that detect these violations that do not fit to pre-defined symptoms of poor design

5. DISCUSSION

The quality of our modularity violation detection approach depends heavily on the availability of modifications requests and their solutions. For small-scale projects or projects without version control systems, it is hard to apply CLIO.

When calculating change coupling, how long a version history is enough? The answer depends on the specific project and how to determine the best threshold is our ongoing work. In the evaluation, we use all available revision histories to determine change couplings. Changing the number of versions used for change coupling may change the results. Our decision of only considering the five most recent releases in evaluation when determining violations is based on the fact that the results do not differ when we consider more versions. Again, this heuristic may vary with different projects.

Since we only applied CLIO to two subject systems, we cannot conclude that the effectiveness of CLIO 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.

Some violations detected using CLIO may not embody any design problems but reveal valid semantic dependency, as shown in previous work [30, 31]. But our experiments show that considerable number of violations indeed reflect design problems. The accuracy of CLIO also depends on how accurate the ACN model embodies design decisions and their assumption relations. The ACN model we used in this paper were automatically generated from UML class diagrams derived from source code. Some dependencies can only be reflected in other design models, such as an architectural description. 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 improve CLIO by using high-level architectural models in addition to reverse-engineered source models.

6. CONCLUSION

Parnas' original definition of a *module* means an independent task assignment, and his information hiding principle advocates separating internal design decisions using an *interface* to allow for independent evolution of other modules. Though this definition of *modularity* is inherently inseparable from the notion of *independent module evolution*, existing approaches do not detect modularity violations by comparing how components should change together and how the components actually change together.

This paper proposes a novel approach of identifying eroding design structure by computing the discrepancies between modularity-based impact scope analysis and change coupling-based impact scope analysis. We evaluated CLIO using the version histories of Hadoop Common and Eclipse JDT. We conservatively confirmed hundreds of reported violations to be correct, assuming there are no other design problems in those code bases except the ones the developers already refactored or reported. The result also shows that detected modularity violations exhibit various symptoms of poor design, showing CLIO's advantages in contrast to bad-code smell detection techniques that find only pre-defined set of poor design symptoms, without regard to the system's original design structure nor its evolution history.

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] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, Aug. 2006.
- [4] Y. Cai and K. J. Sullivan. Modularity analysis of logical design models. In *Proc. 21st ASE*, pages 91–102, Sept. 2006.
- [5] 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.
- [6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *TSE*, 20(6):476–493, June 1994.
- [7] B. Fluri, H. C. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *Proc. 5th SCAM*, pages 66–74, Sept. 2005.
- [8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, July 1999.
- [9] 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.
- [10] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Nov. 1994.
- [11] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Proc. 13th CSMR*, pages 255–258, Mar. 2009.
- [12] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. SIGMOD*, pages 1–12, May 2000.
- [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, and W. Shen. Automatic transformation of UML models into analytical decision models. Technical Report DU-CS-08-01, Drexel University, Apr. 2008.
<https://www.cs.drexel.edu/node/13661>.
- [15] S. Huynh, Y. Cai, Y. Song, and K. Sullivan. Automatic modularity conformance checking. In *Proc. 30th ICSE*, pages 411–420, May 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] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–8, Dec. 1972.
- [19] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *Proc. 2nd MSR*, pages 1–5, May 2005.
- [20] M. P. Robillard. Topology analysis of software dependencies. *TOSEM*, 17(4):18:1–18:36, Aug. 2008.
- [21] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. 20th OOPSLA*, pages 167–176, Oct. 2005.
- [22] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *TSE*, 17(2):141–152, Feb. 1991.
- [23] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *Proc. 8th FSE*, pages 99–108, Sept. 2001.
- [24] 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.
- [25] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities. In *Proc. 13th CSMR*, pages 119–128, Mar. 2009.
- [26] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *TSE*, 35(3):347–367, May 2009.
- [27] S. Wong and Y. Cai. Predicting change impact from logical models. In *Proc. 25th ICSM*, pages 467–470, Sept. 2009.
- [28] 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.
- [29] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A hybrid approach to identify framework evolution. In *Proc. 32nd ICSE*, May 2010.
- [30] 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.
- [31] 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.