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 refactoring costs. Clio computes the
discrepancies between how components should change together
based on the modular structure and how components actually
change together as reflected in version history. 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 exhibit 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

General Terms

design rule theory, 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 [2, 17]. 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 [14]. Such activities cause modularity decay over time and may require expensive system-wide refactoring.

Though empirical studies have revealed a strong correlation
between software defects and modularity violations [6, 21],
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 consistently change together to accommodate modification
requests, but they belong to two separate modules that are
supposed to evolve independently, we consider this as a
modularity violation.

Clio has three components. The first component calculates structural coupling—how components should change
together, based on Baldwin and Clark’s design rule theory
design structure matrix (DSM) [2] modeling. The second
component extracts change coupling—how components actually change together [10] through mining the project’s
revision history. The third component identifies 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 Com-
mon,2 and 10 releases of Eclipse JDT.3 Our evaluation strat-

1Consistent with Ying et al. [30], a modification request can
be a bug fix or feature enhancement. The set of files that
resolve a modification request is called its solution.

2http://hadoop.apache.org/common/

3http://www.eclipse.org/jdt/
sponding code to determine whether the detected violations reveal symptoms of poor design.

We identified 231 modularity violations (47%) from 490 modification requests of Hadoop, of which 152 (65%) violations were confirmed. From 3458 modification request of Eclipse JDT, CLIO identified 399 modularity violations (12%), which shows that the changes in Eclipse better match its modular structure. 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 alerted 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 Code Smells. Fowler [9] 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. [16] presented the Decor tool and domain specific language (DSL) to automate the construction of design defect detection algorithms. Several other approaches [23–25] automatically identify bad smells that indicate needs for refactoring. For example, Tsantalis and Chatzigeorgiou’s technique [24] 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 clone refactoring candidates using structural metrics (e.g., the number of assigned variables, the number of referred variables).

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 deviates from the designed modular structure. Second, by taking version histories as input, CLIO detects the most recently and frequently occurring violations, instead of bad smells detected in a single version without regard to the program’s evolution context. Similar to CLIO, Ratzeing et al. [18] also detect bad smells by examining change coupling. Their approach leaves it to developers to identify violations from the 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. Some of the most widely used design structure matrix (DSM) tools include Lattix, Strute 101, and NDepend. These tools support automatic derivation of DSMs from source code, modeling the syntactic dependencies between classes or files. Different from these tools, the DSMs used in CLIO are generated from augmented constraint networks (ACNs) [4, 5], which separate the interface and implementation of a class into two design dimensions, and manifest implicit and indirect dependencies [26] that cannot be revealed by a syntactical DSM [4].

Sangal et al. [20] describe how to use Lattix to identify undesired dependencies. Using Lattix, a user can specify which classes should not depend on (i.e., syntactically refer to) which other classes. The tool raises an alert if a predefined constraint is violated. A key difference between CLIO’s and Lattix’s detection techniques is that CLIO analyzes version histories to detect violations that occur during software evolution, many of which are not in the form of syntactical dependencies 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 (e.g., Selby and Basili [21]). Various metrics have been proposed (e.g., Chidamber and Kemerer [7]) to measure coupling and failure proneness of components. The relation between change coupling [10] and defects has also been recently studied. Cataldo et al.’s [6] study reveals a strong correlation between density of change coupling and failure proneness. Fluri et al.’s [8] study shows that a large number of change coupling relationships are not entailed by structural dependencies. While the purpose of these studies is to statistically account for the relationship between software defects, change coupling, and syntactic dependencies, CLIO’s purpose is to locate modularity violations that may cause software decay and defects.

3. DETECTION APPROACH

This section presents our modularity violation detection approach, supported by the CLIO framework. Section 3.1 provides an overview and the following subsections elaborate the major components and their background knowledge.

3.1 Framework Overview

Suppose that a number of modification requests (MRs) are fulfilled when a project evolves from version $n$ to $n+1$. Figure 1 depicts how the project manager can use CLIO to determine whether these changes violate the designed modular structure so that modularity decay can be detected.

CLIO employs a plugin architecture and has three major components, along with supporting tools. The first major component, $dr$-predict, computes a set of files that are likely to be changed together according to the designed modular structure (Section 3.3). We leverage the design structure matrix (DSM) model [2], which can be derived from an augmented constraint network (ACN) [4]—a design model based on Baldwin and Clark’s design rule theory [2]. An ACN, in turn, can be derived from source code or design

http://www.lattix.com/

http://www.headwaysoftware.com/products/structure101/

http://www.ndepend.com/

7 CLIO is the Greek muse of history.
models [26,27]. We introduce these background concepts in Section 3.2. Figure 1 shows that our framework provides a tool (Moka [26]) that can reverse-engineer a UML class diagram from Java binaries, and a tool (Janus [26]) that converts the class diagram into an ACN and a DSM to be used by the dr-predict plugin.

The second major component, logic-predict plugin computes the components that are likely to be changed together according to change coupling, derived by the extract plugin. The extract plugin of Clio first records the set of files changed together in transactions\(^8\) of revision history, and stores the support and confidence values of each change coupling into a database, following the work of Ying et al. [30] and Zimmermann et al. [31]. For the solution \(S\) of each modification request, the logic-predict plugin selects a subset of \(S\) that exhibit the strongest change coupling with other files according to the change coupling database. We call this selected set of files the starting change set \(\sigma\).

The logic-predict plugin predicts the change impact of \(\sigma\) as follows: a file is predicted to be in the impact scope of \(\sigma\) if the corresponding association rule’s support and confidence values are above the minimum support \(s_{th}\) and confidence \(c_{th}\) thresholds.\(^9\) The impact scope of \(\sigma\), computed by logic-predict, is noted as FileSet B in Figure 1.

The logic-predict plugin shares the \(\sigma\) with dr-predict so that both plugin components can compute the impact scope of the same set of files. The modularity-based impact scope of \(\sigma\), computed by the dr-predict, is noted as FileSet A in Figure 1. Since \(\sigma\) consists of files that reveal strongest change coupling with other files, the discrepancy between \(\sigma\)'s impact scopes based on structural couplings and change couplings is mostly likely to reveal modularity violations.

Finally, given \(A\) and \(B\), and a MR solution \(S\), the third major component of Clio, the detect plugin, computes a set of discrepancies, \(D = (B \cap S) \setminus A\). By using \(B \cap S\), the detect plugin filters out files that were accidentally changed together. Recurring discrepancies (a subset of files in \(D\)) are then reported to the users as violations.

Since the logic-predict plugin is a reimplementation of existing work, in the following subsections we mainly elaborate the modularity-based impact scope analysis approach embodied by the dr-predict plugin (Section 3.3), the necessary background of it (Section 3.2), as well as the discrepancy calculation method embodied by the detect plugin (Section 3.4).

### 3.2 Background

This section introduces key background concepts of our modularity-based impact scope analysis approach embodied by the dr-predict plugin. We first introduce the augmented constraint work (ACN) model that is used to derive a design structure matrix (DSM). By clustering the DSM into a special form called the design rule hierarchy (DRH), modules,

\(^8\)A 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 (e.g., cus2sm) have been developed to reconstruct transactions.

\(^9\)Consistent with Zimmermann et al. [31], the frequency of a set in a set of transactions \(T\) is \(f rq(T, x) = |\{t \in T | x \subseteq t\}|\). 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 \(conf(T, x_1 \Rightarrow x_2) = \frac{frq(T, x_1 \cup x_2)}{frq(T, x_1)}\).

**Figure 1**: Approach Overview: the Clio Framework

**Figure 2**: Maze game UML class diagram [28]

Augmented Constraint Network (ACN). An ACN consists of a constraint network and a dominance relation. Figure 3 shows part of an ACN derived from the above UML class diagram. The constraint network models design deci-
sions as variables and model their assumption relations as logical constraints. In the maze game example, each class is modeled using two variables (lines 1–6): an interface variable\(^ {10} \) 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).

The dominance relation in an ACN describe asymmetric dependency relationships among design decisions, the essence of Baldwin and Clark’s concept of design rules [2]. Baldwin and Clark coined the 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 (e.g., 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 [22]. 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. 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 prior work [26].

1. MapSite_interface : \{orig, other\}
2. MapSite_impl : \{orig, other\}
3. Room_interface : \{orig, other\}
4. Room_impl : \{orig, other\}
5. Maze_interface : \{orig, other\}
6. Maze_impl : \{orig, other\}
7. Room_impl = orig ⇒ MapSite_interface = orig
8. Room_impl = orig ⇒ Maze_impl = orig
9. Maze_impl = orig ⇒ Room_interface = orig
10. (MapSite_impl, MapSite_interface)
11. (Room_impl, MapSite_interface)
12. (Maze_impl, Room_interface)

![Figure 3: Partial Maze game ACN [28]](image)

**Design Structure Matrix (DSM)**. Figure 4 shows a DSM automatically derived from the 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 pairwise dependency relation defined on ACN: if \(y\) depends on \(x\), it means that \(y\) must be changed in one of multiple ways to restore the ACN consistency that is broken by changes to \(x\), and that \(y\) is not a design rule of \(x\). If so, the cell on row \(y\), column \(x\) will be marked. For example, cell (r11, c2) indicates that Room_impl depends on MapSite_interface.

**Design Rule Hierarchy (DRH)**. In order to identify modules—dependent task assignments according to Parnas’ definition [17], our prior work defined a special clustering based on the 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 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.

Figure 4 shows a DSM clustered into a DRH with four layers (outer rectangle in bold line along the diagonal) 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 Door-NeedingSpell_interface (r12). The modules in the last layer (r14-24, c14-24) can be designed, changed, and replaced concurrently with each other, not 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.

### 3.3 Modularity-based Impact Scope Analysis

Given a starting change set \(\sigma\) and a DRH-clustered DSM, the dr-predict plugin calculates the change impact of \(\sigma\) as follows: all the files that belong to the same module of \(\sigma\) are within its impact scope; if a file belongs to a module that depends on the module of \(\sigma\), then the more dependencies between the modules, the more likely the dependent module is within the impact scope of \(\sigma\); the design rules of \(\sigma\) should never within its impact scope.

We leverage Robillard’s [19] relevant artifact recommendation algorithm, which identifies a subset of nodes in a graph relevant to the initial set of interests based on the graph’s topology. A DRH-clustered DSM can be represented as a directed acyclic graph where each vertex \(v\) 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 \(v\). To demonstrate our 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].

Starting from the starting change set (with shaded background and white text), we assign a weight \( \mu \), 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 Maze_interface, we examine its neighbors, the subordinate decisions that Maze_interface influences, and assign them a weight. While traversing the graph to assign weights, we ignore the starting change set’s design rules because they are supposed to be 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, MazeFactory_impl’s interface and implementation should not be within its impact scope.

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

\[
\mu_0 = \left(1 + \frac{|S_{\text{forward}} \cap S|}{|S_{\text{forward}}|} - \frac{|S_{\text{backward}} \cap S|}{|S_{\text{backward}}|}\right) ^{\alpha}
\]

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. \( \mu \) is a weight and \( \alpha \) is a constant defined to determine the degree of relevancy propagation.\(^{11}\)

To start 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 the next round of weight assignment. We repeat this iterative process 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 \( \theta_d \) (e.g., 0.75) are then recommended as being in the impact scope (depicted with a dashed enclosure).

### 3.4 Discrepancy Analysis

Given the impact scopes of the starting change set \( \sigma \) calculated by \( \text{dr-predict} \) and \( \text{logic-predict} \), the detect plugin of Clio calculates their discrepancies. Because the impact scope results vary with the thresholds selected, our framework automatically chooses the thresholds with best accuracy, measured using the standard \( F_1 \) value from information retrieval. The \( \text{dr-predict} \) plugin varies the minimum weight threshold \( \theta_d \) from 0 to 0.95 in increment of .05 to find a threshold that maximize \( F_1 \). Similarly, the \( \text{logic-predict} \) plugin independently varies the support threshold from 2 to 10 and varies the confidence threshold from 0 to 0.95 in increment of .05 to find the maximum \( F_1 \).

Given the most accurate predictions from \( \text{dr-predict} \) and \( \text{logic-predict} \), the detect plugin computes their discrepancies and identifies recurring discrepancies over multiple versions of the software, 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 DRH, meaning that they should evolve independently from each other. Clio’s \( \text{dr-predict} \) plugin would never report that they are within each other’s impact scope. If the revision history shows that they consistently change together (e.g., due to similar changes to cloned code) Clio would report that there is a modularity violation. Consider another example, since Maze_interface is the design rule of Room_impl, it is normal that Maze_interface changes

\[\text{Figure 4: Maze game DSM [28]}\]
and influences Room_impl along with other dependent components. But Clio’s dr-predict plugin would never predict MapSite_interface to be in the change scope of Room_impl. If the revision history shows that MapSite always changes with Room_impl, 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 violations identified by Clio?** That is, do the 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 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 the future. 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, making designers aware of violations as soon as possible can help 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 violations identified by our approach?** We 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 toolkit 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 used 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 used the cvs2svn\(^{12}\) 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.

\(^{12}\)http://cvs2svn.tigris.org/

<table>
<thead>
<tr>
<th>Subjects</th>
<th>SLOC</th>
<th>#Transactions</th>
<th>#Releases</th>
<th>#MRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse JDT</td>
<td>137K-222K</td>
<td>27806</td>
<td>10</td>
<td>3458</td>
</tr>
<tr>
<td>Hadoop</td>
<td>13K-64K</td>
<td>3001</td>
<td>15</td>
<td>490</td>
</tr>
</tbody>
</table>

#### Table 1: Characteristics of subject programs

The experiments showed that the results do not significantly differ if we aggregate discrepancies over more than five releases.
chose this technique because it has a 5.01% higher precision and iteratively selects the most likely high-level transforma-
set of seed matches based on name similarity, generates can-
headers from both old and new versions respectively, finds a
refactorings at a method-header level. It extracts method-
gram versions as input and detects nine di
fferent types of
spection to confirm a violation.

confirmed
ing a design problem. If so, we call such violation as being confirmed. We use both automated method and manual in-
spection to confirm a violation.
First, we compared the detected violations with refac-
torings that were automatically found by Kim et al.’s API
matching tool [15]. This API matching tool takes two pro-
gram versions as input and detects nine different types of refactorings at a method-header level. It extracts method-
handers from both old and new versions respectively, finds a
set of seed matches based on name similarity, generates can-
didate high-level transformations based on the seed matches,
and iteratively selects the most likely high-level transforma-
tion 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 comparat-
ive 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 pro-
grammers fixed, or at least plan to fix, these reported vi-
oliations through redesign or refactoring activities. For the
rest of the reported violations, we studied the correspond-
ning source code to see whether they include any symptoms
of poor design.

4.3 Results
We analyzed our results by answering the questions pro-
posed 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 con-
irmed 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 number of reported violations: $\frac{|CV|}{|V|}$.

|          | $|V|$ | $|V \cap R|$ | $|V \cap M|$ | $|CV|$ | Pr. |
|----------|------|-------------|-------------|-------|-----|
| Eclipse JDT | 231  | 81          | 71          | 152   | 66% |

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 viola-
tions 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 vi-
olutions, 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 dis-
covered 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.

In-depth Case Study: Hadoop. Now we present an
in-depth study of Hadoop to demonstrate examples of vio-
lations 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 FS-
Namesystem. FSNamesystem depends on FSDirectory.isVa-
idBlock method, but it often changes with FSNamesystem.
An API-level refactoring was identified in release 0.13.0,
showing that the isValidate 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, 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.

**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 reported 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.

**Refactorings that are not 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 be performed for other purposes besides addressing modularity violations.

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 FSFileSystem 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. 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.

### 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 syntactical DSM, there are no symmetric marks to alert 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, a detected 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 clones. A traditional clone detector would likely identify more clones than Clio, but it may be too costly and unnecessary to refactor all of them. Clio highlights the ones that happen recently and frequently, and hence provides more targeted candidates for refactoring.

---

**Figure 7: Timing of Violation Detection (Hadoop)**

**Table 3: Characteristics of the Violations**

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Cyclic</th>
<th>Clone</th>
<th>Inheritance</th>
<th>Coupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse JDT</td>
<td>Hadoop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>18</td>
<td>37</td>
<td>25</td>
<td>66</td>
</tr>
</tbody>
</table>
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 [9]. 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, Clío 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 dependencies. 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 needed to be refactored.

The FSDirectory and FSNamesystem we mentioned earlier is also an example of unnamed coupling. Clío 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 Latitx DSM, the user can mark that FSDirectory should not depend on FSNamesystem so that if FSDirectory explicitly refers to FSNamesystem, Latitx will raise an alarm. However, 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 Clío.

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 used all available revision histories to determine change coupling. Changing the number of versions used for analysis may alter 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 significantly differ when we consider more versions. Again, this heuristic may vary with different projects.

The selection of a starting change set can significantly affect the accuracy of violation detection. We use the most highly coupled elements in a MR solution as the starting change set. However, other heuristics can be used for selecting a starting change set. Identifying such heuristics and seeing how they affect the accuracy of violation detection is an ongoing future work. Automatically recovering the original starting change set of an MR is an active, but immature, area of research (e.g., Antoniol et al. [1]). Such techniques try to reconstruct what developers would have first modified in fulfilling the MR. As these techniques mature, we can evaluate their effectiveness in improving our approach.

Since we only applied Clío to two subject systems, we cannot conclude that the effectiveness of Clío 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. [3] 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 Clío 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 Clío 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 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 Clío by using high-level architectural models in addition to reverse-engineered source models.

6. CONCLUSION

Parnas’s 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. Problems occur if modules that are designed to be inde-
dependent always have to change together. This paper proposes a novel approach of identifying eroding design structure by computing the discrepancies between how components should change together and how they actually change together. We evaluated Clio using the version histories of Hadoop Common and Eclipse JDT. We conservatively confirmed hundreds of reported violations to be correct. 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 National Science Foundation under grants CCF-0916891, CCF-1043810, and DUE-0837665.

8. REFERENCES


