ADvISE: Architectural Decay In Software Evolution

Salima Hassaine*, Yann-Gaël Guéhéneuc§, Sylvie Hamel*, and Giuliano Antoniol§
*DIRO, Université de Montréal, Québec, Canada
Email: {hassaisa,hamelsyl}@iro.umontreal.ca
§DGIGL, École Polytechnique de Montréal, Québec, Canada
Email: {yann-gael.gueheneuc}@polymtl.ca, antoniol@ieee.org

***Abstract*—Software systems evolve continuously, requiring continuous maintenance and development. Consequently, their architecture tends to degrade with time as it becomes less relevant to new, emerging requirements. Therefore, stability or resilience is a primary criterion for evaluating an architecture. In this paper, we propose a quantitative approach to study the evolution of the architecture of object oriented systems over time. In particular, we represent an architecture as a set of triplets (S,R,T), where S and T represent two classes and R is a relationship linking them. We use these triplets as basic unit to measure the stability of an architecture. We show the applicability and usefulness of our approach by studying the evolution of three open source systems: JFreeChart and Xerces-J and Rhino.**

***Index Terms*—Software evolution; Architecture decay; Architecture stability.**

I. Introduction

Software architecture[1] is an important topic of interest in the object-oriented community, because an architecture contains information that ease the communication among programmers and should help develop programs with better quality.

Software systems evolve continuously, requiring continuous maintenance and development. Thus, they undergo changes throughout their lifetimes as features are added and defects are fixed. As these systems are adapted to changing requirements, they may suffer from signs of aging, because their architecture can deviate substantially from the architecture originally designed, *i.e.*, their architecture tends to decay with time and it becomes less adaptable to new, emerging requirements [1], [2]. When evolution occurs in an uncontrolled manner, the systems become more complex over time and thus, harder to maintain [3], [4]. Therefore, methods and tools are needed to evaluate architectural decay to identify signs of software aging to overcome or avoid their negative effects and thus, to keep development costs down.

Several authors [5], [6] suggested stability or resilience as a primary criterion for evaluating an architecture and, thus, they proposed different approaches to analyse the evolution of software architectures [6]–[10]. Most of these previous approaches aim at finding structural changes occurring in long-lived evolving architectures. However, to the best of our knowledge, none of these approaches have been used to study the signs of software aging or architectural decay to propose mitigating methods and tools.

[1]We use the term "architecture" to mean any structural model of a program, *e.g.*, a UML class diagram.

Other authors defined architectural decay as the deviation from an original design, *i.e.*, the violation of architecture caused by the process of evolution [11]–[13]. They suggested that decayed architectures make their systems more prone to defects [14] and that, in some not-so-rare cases, a system architecture and its implementation code must be thrown away because it is too hard to maintain, unless the decay can be stopped before the architecture is completely unworkable [15].

In this paper we propose a novel approach *ADvISE* to investigate some metrics (code decay indicators) on software, that serve as symptoms, risk factors, and predictors of decay, in the context of an evolving architecture. Our approach *ADvISE* aims at analysing the evolution of a software architecture at various levels (classes, triplets[2], and micro-architectures[3]) to provide support for software evolution. Specifically, we analyse the architectural history of three software systems and infer when and how their architectures decayed and whether the decayed architectures are more prone to defects.

The first step in observing architectural decay is to use a diagram matching technique to identify structural changes among versions of architectures. Finding structural changes occurring in long-lived evolving architectures requires the identification of class renamings. Thus, a first contribution of this paper is a set of structure-based and text-based similarities to identify class renamings in evolving architectures. The second step requires to match evolving architectures to identify changes and stable micro-architectures. Thus, the second contribution are a bit-vector and incremental clustering algorithms to perform the matching between several versions of an architecture and find stable micro-architectures, which exist in all versions. The third step consists of using the previously-identified micro-architectures, in proposing metrics to identify the signs of architectural decay. Thus, the third contribution is a set of metrics (code decay indicators) on software, that serve as symptoms and risk factors of decay in the context of an evolving architecture, and thus, predictors of fault proneness. We also perform a quantitative and two qualitative studies, to show the applicability and usefulness of our approach. We apply our approach on three open-source systems: JFreeChart, Rhino and Xerces-J, and answer the following research questions as follows:

[2]We define a "triplet" as $T = (C_{Source}, R, C_{Target})$, where C_{Source} and C_{Target} represent two classes and R is a relationship between them.

[3]We use the term "micro-architecture" to mean any subset of an architecture, *e.g.*, a set of classes and their relationships.

- **RQ1:** What are signs of architectural decay and how can they be tracked down? We studied the graph of architectures evolution for each system, and we showed that these indicators provide us useful insights regarding the signs of software aging.
- **RQ2:** Do stable and unstable micro-architectures have the same risk to be fault prone? We showed that stable micro-architectures, belonging to the original design, are significantly less bug-prone than unstable micro-architectures.

This paper is organised as follows. Section II summarises related work and highlights their limitations. Section III describes our approach. Section IV presents three case studies, and discusses our approach. Finally, Section VI concludes and outlines future work.

II. Related work

Our work relates to three main research directions: architectural decay, architectural diagram evolution, and API evolution. Architectural decay is due to violation of architecture caused by the process of evolution. Existing techniques for architectural diagram evolution aims detecting structural changes between versions of a class diagram. API evolution techniques aims comparing two versions of a framework, in order to find changes caused by renamings.

A. Architectural Decay

Perry *et al.* [11] suggested that architectural decay is due to violation of architecture caused by the process of evolution. Eick *et al.* [16] suggested that a code is decayed if it is more difficult to change than it used to be. Van Gurp *et al.* [14] defined architectural decay as the cumulative, negative effect of changes on the quality of a software system. Hochstein *et al.* [12] defined archietctural decay as the deviation of actual or concrete architecture from planned or conceptual architecture. Williams *et al.* [13] defined architectural decay as the deviation from original design. Parnas [17] suggested that softwares suffer from various aging problems such as increasing complexity, unstructured code, feature overloading, etc. The phenomenon of software aging is the result of software changes.

B. Architectural Diagram Evolution

Antoniol *et al.* [18] proposed an automatic approach, based on IR techniques, to trace, identify, and document evolution discontinuities at class level. Xing and Stroulia [7] presented UMLDiff tool to automatically detect structural changes between versions of a class diagram by comparing the two directed graphs that represent these diagrams. UMLDiff was also used to study design evolution [8] and design-level structural changes to understand phases and styles of evolution [9]. Antoniol *et al.* [19] recovered diagrams from the source code of programs and compared them in subsequent versions by computing the best matching of bipartite graphs using a maximum match algorithm [20]. Nodes in the bipartite graph are the classes of the two versions and the similarity between them is derived from class and attribute/method names by means of string-edit distances. Their approach does not support relationships. Kpodjedo *et al.* [6] proposed an Error Correcting Graph Matching (ECGM) algorithm to study architectural diagram evolution. This algorithm is derived from search-based techniques: given two architectural diagrams D_1 and D_2, the authors aims finding, among the large set of all possible matchings, a solution that is the best *true matching* between classes of D_1 and D_2. Kimelman *et al.* [21] have designed a Bayesian framework to solve the model correspondence problem. They have implemented a matching algorithm based on their framework.

Existing approaches for architectural diagram evolution aim comparing two versions of an architectural diagram in order to study its evolution. However, to the best of our knowledge, none of these approaches have been used to evaluate the architectural decay.

C. API Evolution

Wei *et al.* developed AURA [22], a novel hybrid approach that combines call dependency and text similarity analyses to provide developers with change rules when adapting their programs from one version of a framework to the next. Dagenais *et al.* developed SemDiff [23], a tool that recommends replacements for framework methods that were accessed by a client program and deleted during the evolution of the framework. Schäfer *et al.* [24] mined framework-usage change rules from already-ported instantiations. The three previous approaches compute support and confidence value on call dependency analysis. Godfrey *et al.* [25] presented a semi-automatic hybrid approach to perform origin analysis using text similarity, metrics, and call dependency analyses. Xing and Stroulia [26] developed Diff-CatchUp to analyse textual and structural similarities of UML architectural diagram to recognise API changes. Kim *et al.* [27] presented an automated approach to infer high-level renaming patterns.

The above approaches detect renamings at method level and use text-based similarities. Thus, they cannot detect renamed methods that do not have similar names with their target methods. Call dependency-based approaches provide useful information to identify renamed methods that may not be detected by text-based approaches. However, they cannot detect renamed methods for target methods that are not used in frameworks and linked programs. In this paper, we propose similarity measures to detect renamings at class level. Our approach could also be adapted to detect renamings at method level, which is the aim of our future work.

III. Approach overview

This section presents our approach to compute some metrics for evaluating architectural decay (see Figure 1). We will describe each step of the approach in details below. Our approach consists of five steps. Given two versions of an object-oriented program, it extracts their class diagrams using an existing tool PADL. Second, it identifies class renamings using a combination of structure-based and text-based similarities. Third, it computes the diagram matching between the

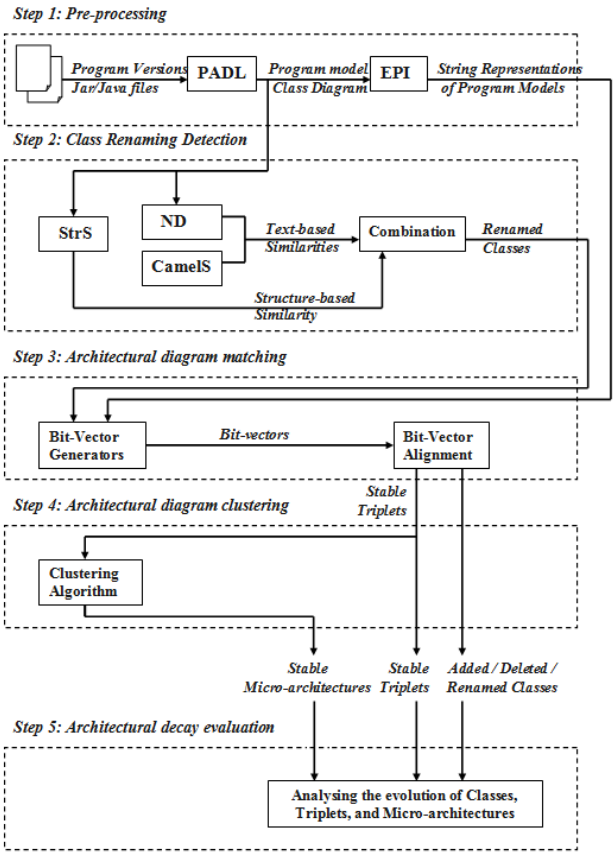


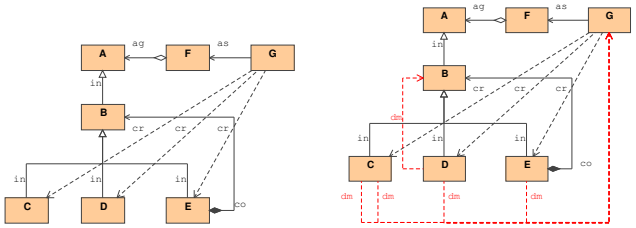
Fig. 1. Approach Overview.

two versions, using a bit-vector algorithm, to identify their common triplets $T = (C_{Source}, R, C_{Target})$, where C_{Source} and C_{Target} represent two classes and R is a relationship linking them. Fourth, it applies an incremental clustering algorithm to group connected triplets into clusters in order to find stable micro-architecture ($S\mu_A$). Finally, our approach reports the sets of $S\mu_A$ between two program versions. We use these sets as an indicator to measure the stability of an architectural diagram.

A. *Step 1: Pre-processing*

We use an existing tool, PADL [28], to automatically reverse-engineer class diagrams from the source code of object-oriented programs. A model of a program is a graph with nodes being the classes and edges representing the relationships between classes, as illustrated in Figure 2(a).

Then, we use an existing tool, EPI [29], to convert the program model into a string representation, defined by the sequence of triplets $T = (C_{Source}, R, C_{Target})$, each triplet representing a connection between the C_{Source} and the C_{Target}. The program conversion consists of three steps. First, EPI takes as input the program model previously generated by PADL. Then, it transforms this graph into Eulerian graph by adding "dummy" edges, noted *dm*, between vertices with unequal in-degree and out-degree (see Figure 2(b)). Finally, traversing the minimum Eulerian circuit generates a unique string representation of the program model (see Figure 2(c)).



(a) UML-like model (b) Eulerian model

A in B in D dm B in E co B in C dm G cr C dm G cr D dm G cr E dm G as F ag A

(c) String representation of the Eulerian model

Fig. 2. Representations of a simple example program (from [29]).

B. *Step 2: Class Renaming Detection*

Based on the program models obtained in the previous step, we first identify class renamings using our structure-based and text-based metrics, which assess the similarities between original and renamed class names.

1) Structure-based Similarity: We define the structure-based similarity (*StrS*), between a candidate renamed class C_A and a target class C_B, as the percentage of their common methods, attribute types, and relationships[4]. We assume that two methods are common in C_A and C_B if they have the same signatures (return types, declaring modules, names, and parameter list). In future work, we plan to use a Levenshtein Distance to assess whether two methods are common or not.

Let $S(C_A)$ and $S(C_B)$ to be the set of methods, attributes, and relationships of C_A (respectively C_B). The structure-based similarity of C_A and C_B is computed by comparing $S(C_A)$ to $S(C_B)$ as:

$$StrS(C_A, C_B) = \frac{2 \times |S(C_A) \cap S(C_B)|}{|S(C_A) \cup S(C_B)|} \in [0,1]$$

If $StrS(C_A, C_B) = 0$, then C_A and C_B do not have common methods, attributes, and relationships. If $StrS(C_A, C_B) = 1$, then classes C_A and C_B have the same sets of methods, attribute types, and relationships. Our algorithm reports, given C_A, the C_B with the highest *StrS* similarity score as the class renamed from C_A.

[4]We compare six types of logical connections: associations, use relations, inheritance relations, creations, aggregations, and container-aggregations (special case of aggregations [30]).

2) Text-based Similarity: It is possible that, our previous algorithm reports, given a candidate renamed class C_A, a set of target classes $\{C_{B_1},...,C_{B_n}\}$ with the highest *StrS* similarity value. In this case, we select the target class that has similar name with C_A. Thus, we compute the text similarity, between a candidate renamed class C_A and each of the target class C_{B_i} $i \in [1,n]$, using a Camel-similarity (*CamelS*), and the Normalized Levenshtein Edit Distance (*ND*).

The *CamelS* similarity between C_A and C_B represents the percentage of their common tokens. We first tokenise the names of C_A and C_B using a Camel Case Splitter. Let $T(C_A)$ and $T(C_B)$ to be the set of tokens of C_A (respectively C_B) names. Then, we compute the *CamelS* similarity of C_A and C_B by comparing $T(C_A)$ to $T(C_B)$ as:

$$CamelS(C_A,C_B) = \frac{2 \times |T(C_A) \cap T(C_B)|}{|T(C_A) \cup T(C_B)|} \in [0,1]$$

If $CamelS(C_A,C_B) = 0$, then the names of C_A and C_B do not have common tokens. If $CamelS(C_A,C_B) = 1$, then the names of C_A and C_B have the same set of tokens.

The Levenshtein Edit Distance [31] between C_A and C_B return the number of edit operations (insertions, deletions, and substitutions) required to transform the name of C_A into the name of C_B. To have comparable Levenshtein distances, we use the normalized edit distance (*ND*), given by:

$$ND(C_A,C_B) = \frac{LEV(C_A,C_B)}{sum(length(C_A),length(C_B)} \in [0,1]$$

where *LEV* computes the Levenshtein distance (we count substitution as an edit operation with cost 1, not as a deletion followed by an insertion with cost 2). If $ND(C_A,C_B) = 0$, then the names of C_A and C_B are the same. If $ND(C_A,C_B) \simeq 1$, then the names of C_A and C_B are different.

Finally, we combine *ND* and *CamelS* to compare the text similarity between names of the candidate class and the target class, because *ND* and *CamelS* assess to two different aspects of string comparison: *ND* is concerned with the difference between strings but cannot tell if they have something in common, while *CamelS* focuses on their common tokens but cannot tell how different they are. Our algorithm reports, the C_B with the highest *CamelS* and the lowest *ND* scores as the class renamed from C_A.

3) Combination of Similarities: We describe our Algorithm 1, as follows: When we compare the similarities of a candidate renamed class C_A to many target classes $\{C_{B_1},...,C_{B_n}\}$, we first compare their structure-based similarity *StrS*. We select the set of target classes having the highest *StrS* value. Then, we compute their textual similarities (*ND* and *CamelS*).

For example, we want to identify a target class that has the most similar name to the candidate renamed class `DataSource`. Let us assume that three target classes, `DataSetdescription`, `DataSet`, and `Dataret` have the highest *StrS* scores (0.70) to `DataSource`. Then, we compute their textual similarities (*ND* and *CamelS*). Both `DataSetdescription` and `DataSet` have the same *CamelS* = 0.50, while their *ND* is different (0.42 for `DataSetdescription`, and 0.29 for `DataSet`). Also, `DataSet` and `Dataret` have the same $ND = 0.29$, while their *CamelS* is different (0.0 for `Dataret` and 0.50 for `DataSet`). Thus, by combining *ND* and *CamelS*, we can identify that `DataSet` has the lowest *ND* and the highest *CamelS* and, thus, is the most similar to `DataSource`.

If we do not find a target class that has the lowest *ND* and the highest *CamelS*. Then, for each target class $\{C_{B_1},...,C_{B_n}\}$, we compare *ND* and *CamelS* similarities to given thresholds. If none of the target classes has *ND* lower than the 0.40 threshold[5] and *CamelS* higher than the 0.50 threshold[6]. Then, we can consider that class C_A was deleted and not renamed.

For example, we want to identify a target class that has the most similar name to the candidate renamed class `BlankAxis`. Let us assume that two target classes, `HorizontalDateAxis` and `HorizontalCategoryAxis` have the highest *StrS* scores (0.66) to `BlankAxis`. Then, we compute their textual similarities (*ND* and *CamelS*). Both `HorizontalDateAxis` and `HorizontalCategoryAxis` have the same *CamelS* = 0.40. However, it is not higher than the 0.50 threshold. Also, their *ND* similarities are not lower than 0.40 threshold (0.44 for `HorizontalDateAxis` and 0.51 for `HorizontalCategoryAxis`). Thus, by comparing *ND* and *CamelS* similarities to given thresholds, we conclude that the original class `BlankAxis` was deleted and not renamed.

C. *Step 3: Architectural Diagram Matching*

A bit-vector algorithm applies a bounded number of vector operations to an input vector, regardless of the length of the input. Such an algorithm can thus be implemented with bit-wise operations available in processors, leading to highly efficient computations [33].

We use a bit-vector algorithm to perform diagram matching of two program versions. This algorithm is interesting, because it can find all common parts in a bounded number of vector operations, which is independent of the size of the diagrams.

We summerise our iterative bit-vector algorithm for architectural diagram matching as follows: we first convert program versions into strings, as described in Section III-A, because bit-vector algorithms are designed for strings. Then, we analyse these strings to identify the set of their common triplets using a bit-vector algorithm [33]. This algorithm consists in traversing in parallel string representations of two the versions, triplet by triplet, and in recording the triplets in the first version that match those in the second version. At the end of the traversal, we obtain the set of stable triplets.

To use a bit-vector algorithm for matching two diagrams, we build the characteristic vectors of each token in the string representation. The characteristic vector of a token *t*

[5]Previous authors [32] have fixed the threshold value of normalized edit distance (*ND*) to 0.40.

[6]We set the 0.5 threshold of *CamelS* similarity through our experimental evaluations on two systems: JFreeChart and Xerces-J.

Algorithm 1 Similarities Combination Principle.

1: $R \leftarrow EmptyList\{\}$
2: $S \leftarrow EmptyList\{\}$
3: $A \leftarrow List\{\{C_{A_1},...,C_{A_n}\}$, candidate renamed classes $(version1)\}$
4: $B \leftarrow List\{\{C_{B_1},...,C_{B_m}\}$, candidate target classes $(version2)\}$
5: **for** each Class C_{A_i} in A, $i \in [1,n]$ **do**
6: **for** each Class C_{B_j} in B, $j \in [1,m]$ **do**
7: Compute Similarity $StrS(C_{A_i},C_{B_j})$.
8: **if** $StrS(C_{A_i},C_{B_j}) > strMax$ **then**
9: $R \leftarrow EmptyList\{\}$.
10: ADD C_{B_j} to R.
11: $strMax \leftarrow StrS(C_{A_i},C_{B_j})$.
12: **else**
13: **if** $StrS(C_{A_i},C_{B_j}) = strMax$ **then**
14: ADD C_{B_j} to R.
15: **end if**
16: **end if**
17: **end for**
18: **for** each Class C_{B_r} in R, $r \in [1,|R|]$ **do**
19: Compute Similarity $CamelS(C_{A_i},C_{B_r})$.
20: Compute Similarity $ND(C_{A_i},C_{B_r})$.
21: **if** $ND(C_{A_i},C_{B_r}) < 0.40$ AND $CamelS(C_{A_i},C_{B_r}) > 0.50$ **then**
22: $ndMin \leftarrow Min(ND(C_{A_i},C_{B_j}),ndMin)$.
23: $camelMax \leftarrow Max(CamelS(C_{A_i},C_{B_j}),camelMax)$.
24: **end if**
25: **end for**
26: $S \leftarrow \{C_{B_s}\}$, having $ndMin$ and $camelMax$.
27: **if** $|S| = 0$ **then**
28: Class C_{A_i} is deleted.
29: **else**
30: Class C_{A_i} is renamed to C_{B_s}.
31: **end if**
32: **end for**

associated with the string $s = s_1...s_m$, is $t = (t_1...t_m)$:

$$t_i = \begin{cases} 1 & \text{if } s_i = t \\ 0 & \text{otherwise.} \end{cases}$$

For the example shown in Figure 2, the characteristic vectors of tokens A, in, and B are defined as:

$$\begin{aligned} A &= \mathbf{1}\underbrace{0000000000000000}_{30}\mathbf{1} \\ in &= 0\mathbf{1}0\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{0000000}_{19} \\ B &= 00\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{00000000}_{20} \end{aligned}$$

We use characteristic vectors to find the common triplets between two versions. Our bit-vector algorithm iteratively reads triplets of tokens in the string representation of first version, *e.g.*, $T = (C_{Source},R,C_{Target})$, and identifies in the string representation of second version—using disjunctions and shifts—all possible triplets that matches T. It stores the triplets identified as being common between two architectural diagram versions.

For example, to identify whether class B is directly related to class A through the inheritance relationship in, we compute:

$$\begin{aligned} (\rightarrow\rightarrow A) &= 0\mathbf{11}\underbrace{000000000000000000}_{29} \\ (\rightarrow in) &= 00\mathbf{1}0\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{000000}_{18} \\ B &= 00\mathbf{1}000\mathbf{1}000\mathbf{1}\underbrace{00000000}_{20} \\ R &= (\rightarrow\rightarrow \mathbf{A}) \wedge (\rightarrow \mathbf{in}) \wedge B \\ &= 00\mathbf{1}\underbrace{000000000000000000}_{29} \end{aligned}$$

and assess whether the bit vector R is null (contains only zeros). If R is not null, then class B is related to class A through the inheritance relationship in and thus, the triplet $T = (A,in,B)$ is common between the two versions.

D. *Step 4: Architectural Diagram Clustering*

Once we obtain the list of all common triplets between two architectural diagram versions using the bit-vector algorithm, we apply our novel incremental clustering algorithm to find the sets of connected triplets, which form the stable micro-architectures ($S\mu_A$) between two program versions.

Our incremental clustering algorithm requires one and only one scan of all triplets. Each triplet is read and then either assigned to one of the $S\mu_A$s or used to start a new $S\mu_A$. Then, the set of existing $S\mu_A$s is reduced by merging two $S\mu_A$s if a new triplet join them.

Algorithm 2 Incremental Clustering Principle.

1: $L \leftarrow EmptyList\{Clusters\}$
2: $S \leftarrow List\{Commontripletsbetweentwoprogramverions\}$
3: **for** each Triplet T in S **do**
4: **for** each Cluster C in L **do**
5: **if** T is in relations with the existing triplet T^* in C **then**
6: **if** T is not added to any cluster **then**
7: ADD T to C.
8: $ClusterToBeMerged \leftarrow C$.
9: **else**
10: MERGE $ClusterToBeMerged$ to C.
11: **end if**
12: **end if**
13: **end for**
14: **if** T is not added to any cluster **then**
15: Create a new Cluster C^*.
16: ADD T to C^*.
17: ADD C^* to L.
18: **end if**
19: **end for**

We describe our Algorithm 2, as follows: Let S to be the list of all common triplets. First, it traverses S, then for each triplet T in S and for each cluster C, if there is a triplet T^* in C in relations with it, then the triplet T is added to the cluster C which is marked as *Cluster to be merged* (lines 3-8). If there is another cluster that contains another triplet T^* in relations with triplet T, then the current cluster is merged with the marked cluster (lines 9-10). If after checking all clusters C in L, the

triplet T was not assigned to any cluster, a new cluster C^* is created and the triplet T is added to it (lines 14-17).

Our algorithm returns the clusters that represent stable micro-architectures $S\mu_A$s between two versions.

E. Step 5: Architectural Decay Evaluation

We perform a pairwise matching of subsequent program architectures to identify the sets of stable triplets and stable micro-architectures. We use these sets as indicators of the architectural stability, as follows:

- Stability of the original design: we compute the number of triplets that has a match in all the versions. These triplets are considered to be part of a tunnel, the backbone part of the system.
- Stability of the architecture with an enriched functionality: we compute the number of triplets that have not changed since their first appearance in a given version of a system.

We will show in Section IV how this indicator can provide useful insights to developers regarding the evaluation of architectural decay in object-oriented programs.

IV. Study Definition and Design

Following the Goal Question Metric (GQM) methodology [34], the *goal* of our study is to analyse the performance of our approach *ADvISE* and understand whether it performs better than previous approaches. The *purpose* is to provide an approach for identifying class renaming and evaluating architectural decay. The *quality focus* is to evaluate the architectural decay of software systems. The *perspective* is of both researchers, who want to study class renaming, and practitioners who analyse software evolution to estimate the effort required for future maintenance tasks. The *context* of our experiments is three open-source Java systems: JFreeChart, Rhino and Xerces-J.

A. Objects

We perform our study using four well-known, open-source programs: JFreeChart, Rhino and Xerces-J. Table I shows some descriptive statistics of these programs.

JFreeChart[7] is a powerful and flexible open-source charting library. Rhino[8] is an open-source implementation of JavaScript written entirely in Java. Xerces-J[9] is a family of software packages for parsing XML.

B. Research Questions

We investigate whether it is possible to apply our approach to study the evolution of architectural diagrams for object oriented software systems. This study aims answering the following research questions:

- **RQ1:** What are signs of architectural decay and how can they be tracked down? This question aims studying if the numbers of stable triplets are good indicators to measure the architectural decay, and if they provide useful insights to developers regarding the signs of software aging.
- **RQ2:** Do stable and unstable micro-architectures have the same risk to be fault prone? This question leads to the following null hypothesis:
  - H_0**:** The proportions of faults carried by stable and unstable micro-architectures are the same.

Program	Releases	Entities (in classes)	Bit-vectors (in bits)	History	Dates
JFreeChart	v0.5.6	1,335	87,227	51	25/11/2000
	v1.0.13	9,105	1,089,345		20/04/2009
Rhino	v1.5.R1	163	40803	11	10/05/1999
	v1.6.R5	449	266,265		19/11/2006
Xerces-J	v1.0.1	5,100	162,583	36	05/11/1999
	v2.9.0	12,585	1,195,353		22/11/2006

TABLE I
STATISTICS FOR THE FIRST AND LAST VERSION OF EACH PROGRAM

C. Analysis Methods

In **RQ1**, we first apply our approach to JFreeChart and Xerces-J. We chose these systems, because the lengths of their histories are long enough to make some interesting observations on the signs of the architectural decay. Also, they are medium-size open-source projects, yet small enough to manually validate the occurrences of class renamings and, studied in previous work. The last condition reduces the bias in the selection of the subject systems and facilitates the comparison with previous work.

Then, we perform a pair by pair matching of subsequent program architectures to identify stable triplets in JFreeChart and Xerces-J (see Figures 3 – 4). To evaluate the deviation of actual architecture from the original design, we compute the number of triplets that has a match in all the versions. These triplets are considered to be part of a tunnel, the backbone part of the system. Also, to analyse the stability of the architecture with an enriched functionality, we compute the number of triplets that have not changed since their first appearance in a given version of a system. Then, we build a graph visualising the evolution of a system architecture over time. The axes of the graphic are the time (software versions) and numbers of our indicators of architectural decay (number of common triplets between two versions, and number of triplets in the tunnel). Thus, we study the graph of architectures evolution for each system to assess whether, these indicators provide us useful insights regarding the signs of software aging.

In **RQ2**, we first apply bit-vector algorithm to identify stable micro-architectures in Rhino. We choose this system, because we use publicly available data on the faults (*i.e.*, issues reporting *real* bugs) collected by previous authors[10]. To attempt rejecting H_0, we test whether the proportion of classes that compose unstable (respectively stable) micro-architectures take part (or not) in significantly more faults than those in stable (respectively unstable) micro-architectures. We use the

[7]http://www.jfree.org/jfreechart/
[8]http://www.mozilla.org/rhino/
[9]http://xerces.apache.org/
[10]http://www.cs.columbia.edu/~eaddy/concerntagger/

contingency tables to assess the direction of the difference, if any. We use Fisher's exact test [35], to check whether the difference is significative. We also compute the odds ratio [35] that indicates the likelihood for an event to occur. The odds ratio is defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the odds that decayed classes are identified as fault-prone, to the odds q of the same event occurring in the other sample, *i.e.*, the odds that stable classes are identified as fault-prone. An odds ratio greater than 1 indicates that the event is more likely in the first sample (unstable classes), while an odds ratio less than 1 that it is more likely in the second sample. An odds ratio $OR = \frac{p/(1-p)}{q/(1-q)}$. $OR = 1$ indicates that fault-prone entities can either have high or low term entropy and context coverage. $OR > 1$ indicates that fault-prone entities indeed have high term entropy and high context coverage. We expect $OR > 1$ and a statistically significant p-value.

V. Empirical Study Results

We now report and discuss the results of our study to address the research questions.

RQ1: Our bit-vector and incremental clustering algorithms identified the common triplets in JFreeChart and Xerces-J. In one hand, we build the graphs visualising the number of triplets between two subsequent versions. In the other hand, we found external information in the release notes and mailing lists. Thus, we report two examples of external information illustrating the architectural changes. The answer **RQ1** is also positive.

1) In JFreeChart: The number of triplets in the first version was decreasing from 413 to 100 stable triplets in the tunnel (see Figure 3). This is due to major changes that have been made. Using external information, we explain the results shown in Figure 3 as follows:

- On Sep 2004, common triplets between versions 0.9.20 and 0.9.21 decreased to 1,894 triplets, the release notes reports an important splitting activity of two packages org.jfree.data and org.jfree.chart.renderer into subpackages category and xycharts. The fully qualified names of all entities in both packages have been changed, which decreased the number of common triplets.
- On Apr 2009, common triplets between versions 1.0.12 and 1.0.13 increased again. After version 1.0.0, we noticed that the number of common triplets was increasing until the last version 1.0.13. The analysis of the release notes reveals that few new features were added and some bugs fixed. The number of common triplets in the tunnel remains constant, the backbone of the program is more stable.

Our approach has the potential to discover two cases of renamings in a fully-qualified class name: (1) Class renaming without changing the package name; (2) Package renaming without changing the class name. In the second case, the triplets are not considered stable, because renamings are due to structural changes in the architecture, such as: splitting or merging packages, moving the class to a new package, etc.

2) In Xerces-J: The number of triplets in the first version was decreasing from 1,693 to 484 triplets have been present and stable in the tunnel throughout all the life of Xerces-J (see Figure 4). This means that 28.58% of the triplets belong to the tunnel. Using external information, we explain the results shown in Figure 4 as follows:

- Between v1.0.1 to v1.4.4, the number of common triplets increased from 1,693 triplets to 3,160: new features were added and maintained until version 1.44. The number of stable triplets in the tunnel decreased, some classes in the first version were deleted and replaced by new ones;
- Between v1.4.4 to v2.0.0, the number of common triplets decreased from 3,160 triplets to 959. There was a major change reported in version 2.0.0. Also, the number of stable triplets in the tunnel decreased from 1,693 to 488, because classes from the first version were deleted;
- Between v2.0.0 to v2.0.9, the number of common triplets increased from 959 triplets to 6,096. The program design became more stable, there was just new features added and some bugs fixed. Also, the number of stable triplets in the tunnel remained constant at 488, so the backbone of the program is now stable;

In JFreeChart and Xerces-J, the number of common triplets between two subsequent versions is increasing over time. As a software evolves, additions of all sorts are to be expected, as new requirements and new functionalities will be implemented in the software. In contrast, deletions are less "natural" and more likely to be associated with the correction of early misconceptions. Thus, for this preliminary study, we are only interested in evaluating how much of an original design is present throughout subsequent versions. To this end, we count the number of triplets in the tunnel of JFreeChart and Xerces-J, to measure their architectural decay[11].

In our approach, a system design is represented by a (possibly reverse-engineered) class diagram. As a result, the first measure of design erosion, we are interested in, is related to how many of the classes of the considered diagram are kept in subsequent snapshots or releases. As illustrated in Figure 3, the tunnel of JFreechart decreased faster than the tunnel of Xerces-J over the n versions, which means the structural changes are more frequent in JFreechart than in Xerces-J. In both programs, numbers of triplets in the tunnel become stable, which means that the systems are more adapted to the new changing requirements.

As a result, the first measure of architectural decay is related to how many of the classes of the considered architecture are kept in subsequent snapshots or releases. Such measure, though unidimensional, is simple, intuitive and can be used for more complex notions, such as estimating a "mortality" rate for classes in a project. The second measure of architectural decay we consider in this study is focused on relationships (number of triplets) existing between classes of a given architecture. The absence of those relationships in subsequent

[11] In our study, we define architectural decay as the deviation of the actual architecture from its original design.

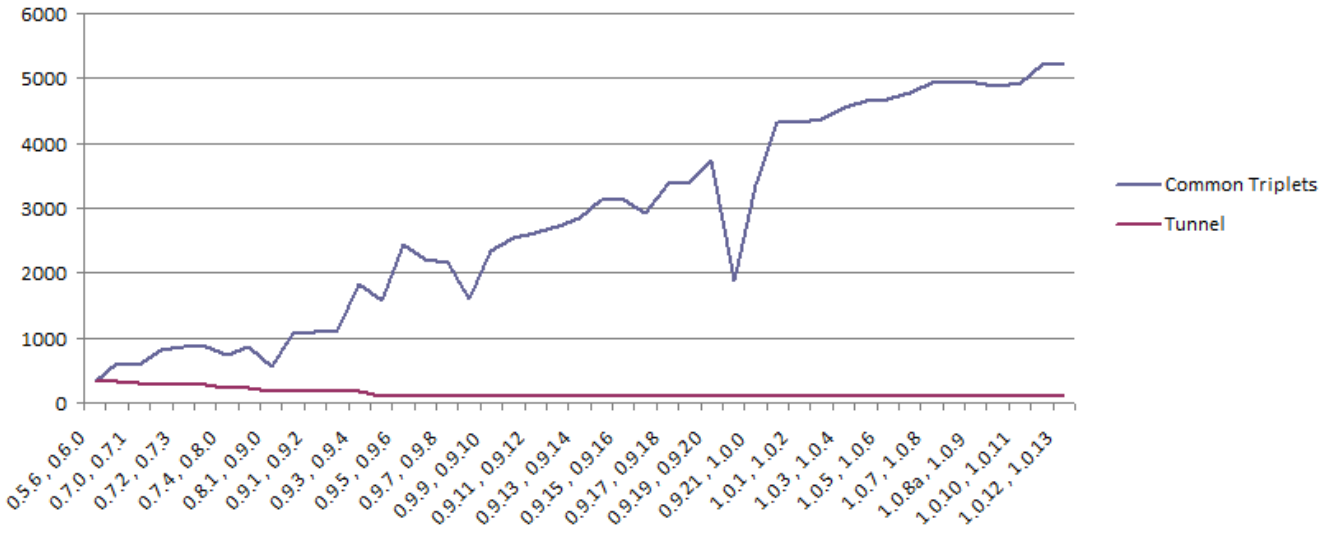


Fig. 3. The evolution of JFreeChart architecture.

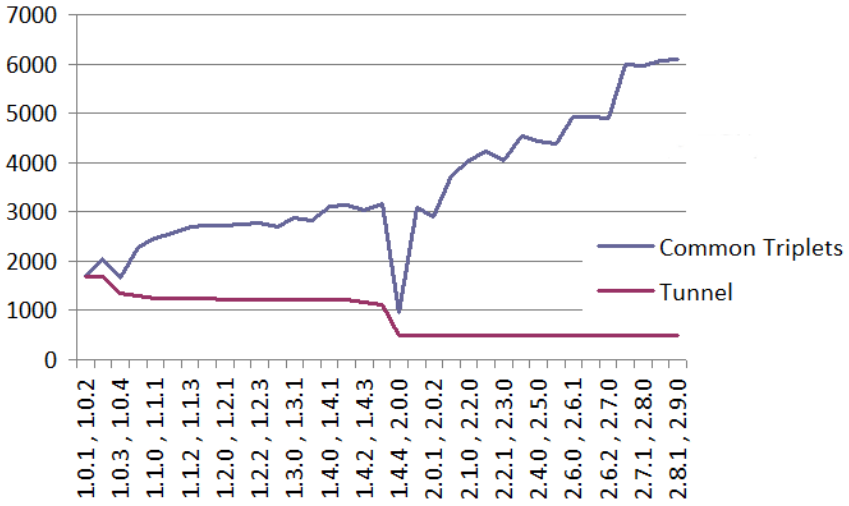


Fig. 4. The evolution of Xerces-J architecture.

versions is another interesting measure of architectural decay. In particular, there may be cases in which the classes are kept but the relationships between them are deeply modified. Finally, our third measure considers the number of connected components of a design in subsequent releases. Considering the set of the classes of a given design, it may happen that the overall connectivity is not preserved. By deleting some relationships (*e.g.*, when trying to insert some new intermediary class), one may add a degree of separation between previously connected classes. In this study, we analysed architectural decay using two measures (number of classes and triplets). In future work, we plan to investigate the use of number of connected components (clusters) to measure the architectural decay.

Time performance: We applied with success and run algorithms on a Linux Bi-Processor Opteron 64-bit with 16 Gb RAM running Redhat Advanced Server version 4. The applicability on the programs JFreeChert and Xerces-J and the computation time—less than 0.036 milliseconds per two consecutive versions (this time does not include the pre-processing step)—is acceptable for an on-line application of the algorithms.

RQ2: Table II presents a 2×2 contengency table for Rhino that reports the number of (1) unstable classes which are identified as fault-prone; (2) unstable classes which are identified as clean; (3) stables classes which are identified as fault-prone; and, (4) stables classes which are identified as clean. The result of Fisher's exact test and odds ratios when testing H_0 are

significants. The p-value is less then 0.05 and the odds ratio for fault-prone unstable classes is three times higher than for fault-prone stable classes.

	Faulty classes	Clean classes
Unstable classes	105	14
Stable classes	39	17
Fisher's test	0.005	
Odd-ratio	3.244	

TABLE II
CONTINGENCY TABLE AND FISHER TEST RESULTS FOR UNSTABLE CLASSES WITH AT LEAST ONE FAULT.

We can answer to **RQ2** as follows: we showed that stable micro-architectures, belonging to the original design, are significantly less bug-prone than unstable micro-architectures.

A. Threats to Validity

Several threats potentially impact the validity of our study.

a) Construct validity: threats concern the relation between theory and observation; in our context, they are mainly due to errors introduced in measurements. Our strategy of reverse engineering class diagrams may contain imprecision and there is need to compare obtained results with other reverse engineering tools. Nevertheless, since all class diagrams were produced by the same tools chain imprecision should factor out. However, we can not exclude the possibility that by using a different reverse engineering tool our algorithms may produce slightly different results. Another critical element is the faults datasets. We use manually-validated faults that have been used in previous studies [12]. Yet, we cannot claim that all fault-prone classes have been correctly tagged or that fault-prone classes have not been missed. There is a level of subjectivity in deciding if an issue reports a fault and in assigning this fault to classes. In this context, we are just interested to check whether a class is faulty or not, rather than quantifying the amount of faults, which is however possible and could be investigate in future work.

b) Internal validity: threats do not affect this particular study, being an exploratory study. Thus, we cannot claim causation, but just relate unstable classes with the occurrences of faults, although our discussion tries to explain why some unstable could have been subject to faults. Conclusion validity threats concern the relation between the treatment and the outcome.

c) Conclusion validity: threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the statistical tests that we used (we mainly used non-parametric tests).

[12]http://www.cs.columbia.edu/~eaddy/concerntagger/

d) External Validity: The external validity of a study relates to the extent to which we can generalise its results. The main threat to the external validity of our study that could affect the generalisation of the presented results relates to the analysed programs. We performed our study on three different Java programs belonging to different domains and with different sizes. However, we cannot assert that our results can be generalised to other larger programs and programs in other programming languages. Future work includes replicating this study on other programs to confirm our results.

VI. CONCLUSION AND FUTURE WORK

Architectural decay is defined as the deviation from an original design, *i.e.*, the violation of architecture caused by the process of evolution [11]–[13]. When evolution occurs in an uncontrolled manner, the systems become more complex over time and thus, harder to maintain [3], [4]. Thus, decayed architectures make their systems more prone to defects [14].

In this paper, we propose a novel approach *ADvISE* and a set of measures to evaluate architectural decay. The first step in observing architectural decay is to use a diagram matching technique to identify structural changes among versions of architectures. Finding structural changes occurring in long-lived evolving architectures requires the identification of class renamings. The second step requires to match evolving architectures to identify changes and stable micro-architectures. The third step consists of, using the previously-identified micro-architectures, in proposing metrics to identify the signs of architectural decay. Thus, this paper present three contributions:

1) The first contribution of this paper is a set of structure-based and text-based similarities to identify class renamings in evolving architectures.
2) The second contribution are a bit-vector and incremental clustering algorithms to perform the matching between several versions of an architecture and find stable micro-architectures, which exist in all versions.
3) The third contribution is a set of metrics (code decay indicators) on software, that serve as symptoms and risk factors of decay in the context of an evolving architecture, and thus, predictors of fault proneness. We also perform a quantitative and two qualitative studies, to show the applicability and usefulness of our approach.

We apply our approach on three open-source systems: JFreeChart, Rhino and Xerces-J, and answer the following research questions as follows:

- **RQ1:** What are signs of architectural decay and how can they be tracked down? We studied the graph of architectures evolution for each system, and we showed that these indicators provide us useful insights regarding the signs of software aging.
- **RQ2:** Do stable and unstable micro-architectures have the same risk to be fault prone? We showed that stable micro-architectures, belonging to the original design, are significantly less bug-prone than decayed micro-architectures.

In future work, we plan to apply our approach to other larger programs and study its results. We also plan to investigate the use of number of connected components (clusters) to measure the architectural decay.

ACKNOWLEDGMENT

This research was partially supported by NSERC (Research Chairs in Software Patterns and Patterns of Software and in Software Evolution) and FQRNT.

REFERENCES

[1] S. J. Carrière and R. Kazman, "The perils of reconstructing architectures," in *Proceedings of the third international workshop on Software architecture*. New York, NY, USA: ACM, 1998, pp. 13–16.

[2] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *J. Syst. Softw.*, vol. 61, pp. 105–119, March 2002.

[3] D. Bell, *Software Engineering, A Programming Approach*. Addison-Wesley, 2000.

[4] D. Hamlet and J. Maybee, *The Engineering of Software*. Addison-Wesley, 2001.

[5] M. Jazayeri, "On architectural stability and evolution," in *da-Europe '02: Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*. London, UK: Springer-Verlag, 2002, pp. 13–23.

[6] S. Kpodjedo, F. Ricca, P. Galinier, and G. Antoniol, "Recovering the evolution stable part using an ecgm algorithm: Is there a tunnel in mozilla?" in *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 179–188.

[7] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 54–65.

[8] ——, "Understanding class evolution in object-oriented software," in *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, june 2004, pp. 34 – 43.

[9] Z. Xing, "Analyzing the evolutionary history of the logical design of object-oriented software," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 850–868, 2005, member-Stroulia, Eleni.

[10] D. Kimelman, M. Kimelman, D. Mandelin, and D. Yellin, "Bayesian approaches to matching architectural diagrams," *Software Engineering, IEEE Transactions on*, vol. 36, no. 2, pp. 248 –274, 2010.

[11] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, pp. 40–52, 1992.

[12] L. Hochstein and M. Lindvall, "Combating architectural degeneration: a survey," *Information Software Technology*, vol. 47, pp. 643–656, 2005.

[13] B. Williams and J. Carver, "Characterizing software architecture changes: An initial study," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, 2007, pp. 410 –419.

[14] J. van Gurp, S. Brinkkemper, and J. Bosch, "Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles," *J. Softw. Maint. Evol.*, vol. 17, pp. 277–306, 2005.

[15] M. W. Godfrey and E. H. S. Lee, "Secrets from the monster: Extracting Mozilla's software architecture," in *Proc. of the Second Intl. Symposium on Constructing Software Engineering Tools (CoSET-00)*, 2000.

[16] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *Software Engineering, IEEE Transactions on*, vol. 27, no. 1, pp. 1 –12, 2001.

[17] D. L. Parnas, "Software aging," in *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press, 1994, pp. 279–287.

[18] G. Antoniol, M. D. Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," *Principles of Software Evolution, International Workshop on*, vol. 0, pp. 31–40, 2004.

[19] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Maintaining traceability links during object-oriented software evolution," *Softw. Pract. Exper.*, vol. 31, no. 4, pp. 331–355, 2001.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 1990.

[21] D. Kimelman, M. Kimelman, D. Mandelin, and D. Yellin, "Bayesian approaches to matching architectural diagrams," *Software Engineering, IEEE Transactions on*, vol. 36, no. 2, pp. 248 –274, 2010.

[22] W. Wu, Y.-G. Guhneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds., vol. 1. ACM, 2010, pp. 325–334.

[23] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 481–490.

[24] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 471–480.

[25] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 166–181, 2005.

[26] Z. Xing and E. Stroulia, "API-evolution support with diff-CatchUp," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 33, no. 12, pp. 818 – 836, 2007.

[27] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, Not Available 2007, pp. 333–343.

[28] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multi-layered framework for design pattern identification," *Transactions on Software Engineering (TSE)*, vol. 34, no. 5, pp. 667–684, September 2008, 18 pages. [Online]. Available: http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/TSE08.doc.pdf

[29] K. Olivier, G. Yann-Gaël, and S. Hamel, "Efficient identification of design patterns with bit-vector algorithm," *csmr*, vol. 0, pp. 175–184, 2006.

[30] Y.-G. Guéhéneuc and H. Albin-Amiot, "Recovering binary class relationships: Putting icing on the UML cake," in *Proceedings of the* 19^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, D. C. Schmidt, Ed. ACM Press, October 2004, pp. 301–314, 14 pages.

[31] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Cybernetics and Control Theory*, vol. 10, no. 8, pp. 707–710, 1966.

[32] L. M. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of identifier renamings," in *Proceeding of the 8th working conference on Mining software repositories*. ACM, 2011, pp. 33–42.

[33] A. Bergeron and S. Hamel, "Vector algorithms for approximate string matching," *International Journal of Foundations of Computer Science*, vol. 13, no. 1, pp. 53–65, 2002.

[34] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Trans. Software Eng.*, vol. 10, no. 6, pp. 728–738, 1984.

[35] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2007.