# On the Relationship Between Program Evolution and Fault-proneness: An Empirical Study

Fehmi Jaafar[1,2], Salima Hassaine[1,2], Yann-Gaël Guéhéneuc[1], Sylvie Hamel[2] and Bram Adams[3]
[1] PTIDEJ Team, École Polytechnique de Montréal, QC, Canada
[2] LBIT Team, DIRO, Université de Montréal, QC, Canada
[3] MCIS Team, École Polytechnique de Montréal, QC, Canada
E-Mails: {jaafarfe, hassaisa, hamelsyl}@iro.umontreal.ca, {yann-gael.gueheneuc, bram.adams}@polymtl.ca

***Abstract*—Over the years, many researchers have studied the evolution and maintenance of object-oriented source code in order to understand the possibly costly erosion of the software. However, many studies thus far did not link the evolution of classes to faults. Since (1) some classes evolve independently, other classes have to do it together with others (co-evolution); and (2) not all classes are meant to last forever, but some are meant for experimentation or to try out an idea that was then dropped or modified. In this paper, we group classes based on their evolution to infer their lifetime models and co-evolution trends. Then, we link each group's evolution to faults. We create phylogenetic trees showing the evolutionary history of programs and we use such trees to facilitate spotting the program code decay. We perform an empirical study, on three open-source programs: ArgoUML, JFreechart, and XercesJ, to examine the relation between the evolution of object-oriented source code at class level and fault-proneness. Our results indicate that (1) classes having a specific lifetime model are significantly less fault-prone than other classes and (2) faults fixed by maintaining co-evolved classes are significantly more frequent than faults fixed using not co-evolved classes.**

***Keywords*—Evolutionary history; co-evolution; reverse engineering; fault-proneness; bit vectors**

## I. Introduction

Object-oriented programs evolve continuously, requiring constant maintenance and development [1]. Thus, they undergo changes throughout their lifetimes as features are added and faults are fixed. When evolution occurs in an uncontrolled manner, the programs become more complex over time and thus, harder to maintain [2][3]. At the same time, the software architectures tend to degrade with time as they become less relevant to new and emerging requirements. This design decay can be detected by measuring the instability of the program artefacts [4], high fault rates [5], and poor code quality [5][6].

For example, Ostrand et al. [7] found that 20% of classes contains 80% of faults. At the same time, these 20% of classes accounted for 50% of the source code. Assuming that all classes are considered to have the same likelihood for fault-proneness is not realistic, because, for example, not all classes are there to last forever, some are meant for experimentation, so it could be expected that they have more faults.

Several fault prediction approaches were proposed to analyse fault-proneness. While some approaches predict the presence or absence of faults for each component (the classification scenario), others predict the amount of faults affecting each component in the future, producing a ranked list of components. On the one hand, Change-Log Approaches [8] use process metrics extracted from the versioning system, assuming that recently or frequently changed classes are the most probable source of faults. On the other hand, Code-Metrics approaches [9] use source code metrics, assuming that complex or larger classes are more fault-prone. However, it is not clear how classes with different evolution behavior are linked with faults. Indeed, evolution studies out there did not link different evolution behavior to faults.

Two major kinds of class evolution are class lifetime and class co-evolution. For example, in ArgoUML, we spotted that hundreds of classes existed only during some program versions. We found that some of these classes, such as `GoModelToCollaboration` and `UMLInstance-ClassifierListModel`, were created by developers to examine a feature which was later abandoned. We differentiate between classes that appear and disappear many times during the program lifetime, (Transient classes) and classes that appear only during one version of the program (Short-lived classes). Similarly, distinguishing between co-evolving classes (classes which exhibit similar evolution profiles, due to interdependencies among them [10]) and independently evolving classes could make a difference. For example, In XercesJ, we found that the class `Document-Builder` co-evolved with the class `SAXParser.java` from Xerces1.0.1 to Xerces2.0.0. Indeed, these two classes are related to the same fault fixed on 26th September 2002.

To evaluate the fault-proneness of these different types of classes, this paper proposes a novel approach, Profilo, to analyse program evolution and fault-proneness. Our approach identifies co-evolution relations among classes and groups classes based on their lifetimes to infer their evolution and co-evolution trends.

We apply our approach on three open-source programs: ArgoUML, JFreeChart, and XercesJ to answer the following

research questions:

- **RQ1**: What is the relation between class lifetime and fault-proneness?
  We decided to consider three types of class evolution: Persistent, Short-lived, and Transient classes. We showed that Persistent classes are significantly less fault-prone than Short-lived and Transient classes.
- **RQ2**: What is the relation between class co-evolution and fault-proneness?
  We found that, in most cases, fixing faults in class A requires changing the co-evolved classes of A.

This paper is organised as follows: Section II presents a pilot study to assess the purpose of analysing the relation between program evolution and fault-proneness. Section III presents our approach Profilo. Section IV describes our empirical study. Section VI, Section V and Section VII report and discuss its results as well as threats to its validity. Section VIII relates our study with previous work. Section IX concludes with future work.

## II. Pilot Study

To motivate the need for studying class evolution for different groups, we discuss a pilot study in which we use phylogenetic trees [11] to observe the impact of fixing faults on program evolution.

Phylogenics is the study of how organisms relate to one another and can be ordered: a phylogenetic tree is a reconstruction of the evolutionary history of species [12]. We create phylogenetic trees for programs to identify how their versions are related from a historical perspective, *i.e.*, they show commonalities, divergences, and evolution trends in order to help developers to understand the full complexity of programs. These trees help developers in understanding and possible detecting design decay by spotting major changes between program versions. The creation of phylogenetic trees is explained in details in Section III.

For example, in JFreeChart, we analysed 46 different versions. To simplify the trees, we use a simplified name for version "0" to version "N" in the phylogenetic tree. This tree, see Figure 1, shows a global pattern of amendments (change in the architecture of the program, changes in classes implemented and evaluated in versions, etc.) in the code of this program before the publication of the version34 (jfreechart-1.0.2) in August 2006 and in the period of publication of the version37 (jfreechart-1.0.5) in March 2007. On the one hand, when we analyse the evolution of fixed faults in JFreeChart, see Figure 2, we noted that these periods correspond exactly to the period with the highest number of faults found and fixed for this project.

On the other hand, the majority of Transient and Short-lived classes in JFreeChart were added in this period (more than 70% of Transient and Short-lived classes on this program). We suspect a correlation, between the introducing of these classes in the program and the increasing number of faults on this period. At the same time, some of this classes are added, renamed, and changed on the same version over their whole lifespan. We found that these classes have similar evolution trends and that many of them are involved in the same faults. Detecting dependencies of evolution of these classes could explain and possibly prevent faults by being sure that changes are propagated adequately by developers among them.

Figure 1. Phylogenetic tree of JFreeChart

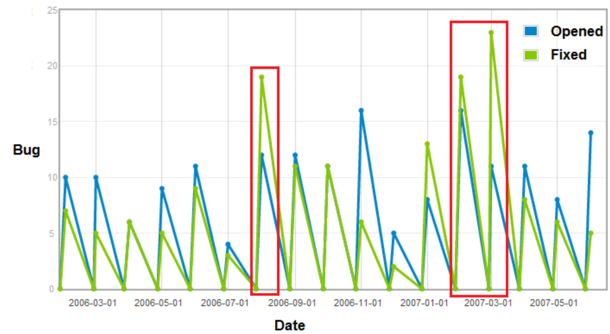


Figure 2. The evolution of fixed and opened bugs JFreeChart

We present a novel approach, Profilo, to (1) group classes in an object-oriented programs according to their evolutionary histories, (2) spot their co-evolution profiles, and (3) relate their evolution and co-evolution trends with fault-proneness. Our goal is to understand amendments and decay in the code of programs and to spot the impact of maintenance activities and program evolution on fault-proneness.

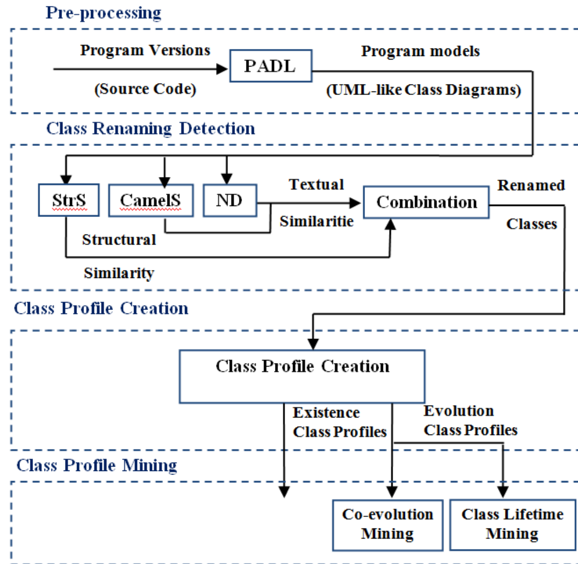


Figure 3. Approach Overview

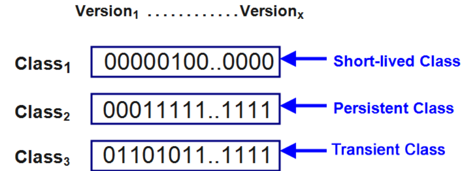


Figure 4. Types of class evolution.

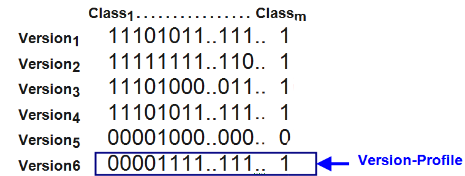


Figure 5. Version-profiles and Existence Class-profiles

## III. Approach Overview

This section presents our approach, Profilo, to analyse the link between program evolution and fault proneness. We will describe each step of the approach in details below as shown in Figure 3. Given several versions of an object-oriented program, Profilo extracts their class diagrams using an existing tool PADL[1]. Profilo creates the set of version-profiles that spots for each version all of its classes. Then, it creates a phylogenetic tree that describes the evolution of different versions. To investigate the relationship between the evolution of programs modeled by phylogenetic trees and the code decay indicated by fault-proneness, Profilo identifies class renamings, class changes, and fault fixing using previous approaches: ADvISE [4] and Macocha [13]. Profilo creates the set of class-profiles that describes the evolution of each class in the program. Based on this set, it groups classes according to their co-evolution relations.

### A. Step 1: Pre-processing

We use PADL [14] to automatically reverse-engineer class diagrams from the source code of object-oriented programs[2] and Macocha to identify the set of changes performed on each class by mining version-control systems. We compute the fault-proneness of a class by relating fault reports and commits to the class. Fault fixing changes are documented in text reports that describe different kinds of problems in a program. Thus, we trace faults/issues to changes by matching their IDs and their dates in the commits and in the bug reports. For example, we detect around three thousands classes in JFreeChart and we trace 420 faults.

### B. Step 2: Version-profile Creation

We define a version-profile for each version of a program, as a vector $v = v_1...v_n$, where $n$ represents the number of classes *i.e.*, the union of the classes in all versions. The value of $v_j$ indicates whether the $j^{th}$ class is present or not in a version (see Figure 5).

$$v_j = \begin{cases} 1 & \text{if class } j \text{ is present or renamed} \\ 0 & \text{otherwise.} \end{cases}$$

The version-profiles are useful for creating the phylogenetic tree of the program.

### C. Step 3: Phylogenetic Tree Creation

A phylogenetic tree [16] is a model created from a set of version-profiles, showing the inferred evolutionary relationships among various versions based upon their similarities and differences. We use the phylogenetic trees to assess more accurately the contributions of the versions relative to the others and to show the decay of the program across versions. We use two existing tools PHYLIP[3] and Phylodendron[4] to draw the phylogenetic tree of a program. Concretely, we use the Hamming distance to measure the amount of differences between two version-profiles, *i.e.*, the number of positions at which the corresponding bits are different. Then, we construct a phylogenetic tree that places related versions under the same interior node and whose branch lengths reproduce the observed distances between versions. Figure 1 illustrates a phylogenetic tree representing the evolution of 46 versions of JFreeChart from the first published version in December 2000 to version 1.0.14 in November 2011. To explain the amendments in the code of this program

[1] http://www.ptidej.net/tool/

[2] We consider six types of static relationships among classes: associations, use relations, inheritance relations, creations, aggregations, and container-aggregations (special case of aggregations [15])

[3] PHYLIP is a free package of programs for inferring phylogenies. http://evolution.genetics.washington.edu/phylip.html

[4] Phylodendron is a free program for drawing phylogenetic trees. http://iubio.bio.indiana.edu/treeapp/treeprint-form.html

discussed in Section II, we decide to study the evolution of its classes and to analyse the fault-proneness of object-oriented source code at class level.

### D. Step 4: Class Renaming Detection

ADvISE identifies class renamings using the structure-based and the text-based metrics, which assess the similarities between original and renamed classes, as follows:

*1) Structure-based and Text-based Similarities:* The structure-based similarity ($StrS$), between a candidate renamed class $C_A$ and a target class $C_B$, is defined as the percentage of their common methods, attribute types, and relationships. 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. The Normalized Edit Distance ($ND$) between $C_A$ and $C_B$ is defined as:

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

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 and the text-based similarities of $C_A$ and $C_B$ is computed by comparing $S(C_A)$ to $S(C_B)$ using the Jaccard index of similarity [17]. 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 $CamelS$).

### E. Step 5: Class-profiles Creation

Profilo mines source code and version control systems to create a class-profile for each class, as follows:

**Evolution Class-profile:** We use this class-profile to extract the co-evolution relations among classes. It is defined as a vector $y = y_1...y_m$, where $m$ represents the number of versions. The value of $y_i$ indicates whether the class $C$ is present, renamed, changed, or deleted in the $i^{th}$ version.

$$y_i = \begin{cases} 2 & \text{if class is renamed or changed at version } i \\ 1 & \text{if class is present at version } i \\ 0 & \text{otherwise.} \end{cases}$$

The co-evolution relations are useful for identifying the sets of classes that evolve together.

### F. Step 6: Class-profiles Mining

*1) Mining Class Lifetime:* We classify classes according to their class-profiles. Then, Profilo reports three types of class evolution as shown in Figure 4.

**Short-lived classes:** They have a very short lifetime, *i.e.*, they exist only during one version of the program. Such classes may have been created to try out an idea that was then dropped or modified.

**Persistent classes:** They never disappear after their first introduction into the program. On the one hand, Persistent classes should be examined, as they may represent cases of dead code that no developer dares to remove as there is no one being able to explain the purpose of these classes. On the other hand, Persistent classes may be considered to be part of a tunnel [18], the backbone part of the program, as they have not been removed since their first appearance in a given version of a program. Hence, we also mine version control systems to assess whether a Persistent class is dead code or not.

**Transient classes:** They appear and disappear many times during the program lifetime. Such classes may have been involved in many design choices and should be analysed, as they represent cases of design decision changes.

*2) Mining Co-evolution Relations:* We group classes that have the same Evolution Class-profile and are related by static relationships. Such classes are added, renamed, changed, and could be deleted in the same versions. They are related, also, by static relationships (use, association, aggregation, and composition relationships).

## IV. Empirical Study

Following the Goal Question Metric (GQM) [19], the *goal* of this study is (1) to detect interesting observations on the relationship between the evolution of object-oriented source code at class level and fault-proneness, (2) to detect co-evolution dependencies to explain and possibly prevent faults, and (3) to confirm these observations statistically. The *quality focus* is the reduction of comprehension cost and maintenance effort. The *perspective* is of both researchers, who want to study the relationship between program evolution and fault-proneness, and practitioners, who analyse software evolution to estimate the effort required for future maintenance tasks. The *context* of our experiment is three open-source Java programs: ArgoUML, JFreeChart, and XercesJ.

### A. Objects

We apply our approach on three Java programs: ArgoUML[5], JFreeChart[6], and XercesJ[7]. We use these programs because they are open source, have been used in previous work, are of different domains, span several years and versions, and underwent between thousands and hundreds of classes. Table I summarises some statistics about these programs.

ArgoUML is an UML diagramming program written in Java. We analyse the evolution of this program for a period of nine years, from 2002-10-09 to 2011-04-03. In this period,

[5] http://argouml.tigris.org/
[6] http://www.jfree.org/
[7] http://xerces.apache.org/xerces-j/

Table I
DESCRIPTIVE STATISTICS OF THE OBJECT PROGRAMS

| | ArgoUML | JFreeChart | XercesJ |
|---|---|---|---|
| Versions | 18 | 46 | 36 |
| Start study | 02-10-09 | 00-12-01 | 03-10-13 |
| End study | 11-04-03 | 11-11-20 | 06-11-23 |
| From Version | 0.10.1 | 0.5.6 | 1.0.1 |
| To Version | 0.32.2 | 1.0.13 | 2.9.0 |
| # of classes | 2011 | 1938 | 892 |

Table II
CARDINALITIES OF THE SETS OBTAINED IN THE STUDY

| | ArgoUML | JFreeChart | XercesJ |
|---|---|---|---|
| Transient | 690 | 645 | 313 |
| Persistent | 1241 | 1293 | 537 |
| Short-lived | 80 | 324 | 42 |
| # of Co-Evolution | 42 | 11 | 23 |

ArgoUML has gone through over 18 major versions, from the version 0.10.1, to the version 0.32.2.

JFreeChart is a Java open-source framework to create complex charts. We analyse the evolution of this program for a period of 10 years. In this period, JFreeChart has gone through 46 major versions, from the first published version on December 2000 to version 1.0.14 on November 2011.

XercesJ is a collection of software libraries for parsing, validating, and manipulating XML. We analyse the evolution of this program for a period of three years, from 2003-10-13 to 2006-11-23. In this period, XercesJ has gone through 36 major versions.

Fault-proneness refers to whether a class underwent at least a fault fixing change during the study periods. Fault fixing changes are documented in text reports that describe different kinds of problems in a program. They are usually posted in issue-tracking systems *e.g.*, Bugzilla, for the three studied programs by users and developers to warn their community of pending issues with its functionalities; issues in these systems deal with different kinds of change requests: fixing faults, restructuring, and so on. We trace faults/issues to changes by matching their IDs in the commits and by manually validation.

## V. EXPLORATORY STUDY

In essence exploratory studies are undertaken to better comprehend the link between program evolution and fault-proneness since very few studies might have been considered in that area.

We use data collected in the three programs and from externals information to discuss typical examples as follows:

**Persistent classes:** In Figure 6, we note that most classes in ArgoUML, JFreeChat, and XercesJ are Persistent (more than 60% of classes). On the one hand, these classes represent the stable backbone (tunnel) of the program such as `org.argouml.uml.generator.ui.ClassGenerationDialog` in ArgoUML. In fact, this class implement the java code generator in this program and was maintained 82 times by several developers (tfmorris, penyaskito, mvw, etc.). On the other hand, Persistent classes could represent also dead code, such as `SDNotationSettings`. Indeed, this class was never changed after its introduction in ArgoUML on March 1999 by tfmorris. We noted that in ArgoUML, more than 80% of classes were maintained three times at most. On the other hand, less than 1% of classes were maintained 50 times at least.

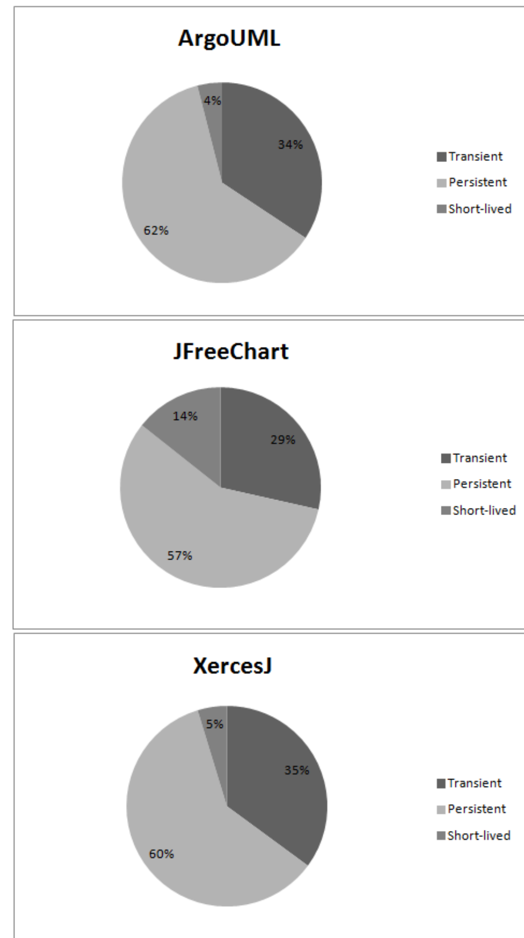


Figure 6. Distribution of class lifetimes detected by Profilo

**Transient classes:** We detect classes that appear and disappear many times during the maintenance of the three programs. For example, the class `OverlaidCategoryPlot`, appeared in JFreeChart in the version 0.9.9 in June 2003, and was deleted in the version 0.20.0 before reappeared in next versions. In fact, developer detect faults in this class, and that explain the Nonpersistence of this class. For example, in the Bugzilla of JFreeChart, the bug ID576760[8] report in relation with this class that "No outline for overlaid category plot" when developers used category plots in one application.

[8]http://sourceforge.net/tracker/index.php?func=detail&aid=576760&group_id=15494&atid=115494

**Short-lived classes:** They represent the smallest group of classes in the three analysed programs. Such classes were created to try out an idea that was then dropped or modified, or to test some program behavior. For example, the class `org.jfree.chart.demo.TimePeriodToStringTest` was created in JFreeChart0.9.9 published in July 2003 to test information encapsulated in `TimePeriod` in order to fix a fault[9] related to this class. After this version, this class was deleted.

**Co-evolution:** The development and maintenance of a program involves handling a large number of classes. Knowing that two or more classes follow the same co-evolution pattern helps developers to maintain properly the dependencies between these classes in the program. Otherwise, they lead to faults in the program. For example, in JFreeChart, we find that `ChartPanel` and `CombinedDomainXYPlot`. were introduced, changed and renamed in the same versions but in different periods and by different developers. Thus, co-change analysis cannot report their dependency. Profilo report that these two files co-evolved and the bugID1950037[10] reported " a bug either in ChartPanel or CombinedDomainXYPlot when trying to zoom in/out on the range axis" and confirmed the dependency between these two classes.

## VI. STUDY RESULTS

Table II summarises the results obtained by applying Profilo. We validated Profilo results manually and checked external sources of information provided by bugs reports, mailing lists, and requirement descriptions to confirm and to discuss results. The analyses reported in this section have been performed using the R statistical environment[11]. We use the contingency tables to assess the direction of the difference, if any.

### *A. What is the relation between class lifetime and fault-proneness?*

*1) Motivation:* We group classes according to their profiles through the program lifespan, taking into consideration the renaming, refactoring, and structural changes of classes, to determine how class lifetime are related to fault-proneness.

*2) Method:* We use Fisher's exact test [20] to check whether the difference is significative. We also compute the odds ratio [20] 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 Short-lived and Transient classes are identified as fault-prone, to the odds $q$ of the same event occurring in the other sample, *i.e.*, the odds that Persistent classes are identified as fault-prone. An odds ratio greater than 1 indicates that the event is more likely in the first sample, 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 (the condition or event under study is equally likely to occur in both groups). $OR > 1$ indicates that fault-prone entities have high term entropy and high context coverage. We expect $OR > 1$ and a statistically significant $p$-value.

We verify the null hypothesis that we state as:

- $H_{RQ1_0}$: There is a statistically significant difference between proportions of faults carried by Persistent, Short-lived, and Transient classes in ArgoUML, JFreeChart, and XercesJ.

If we reject the null hypothesis $H_{RQ1_0}$, then we explain the rejection either as:

- $H_{RQ1_1}$: There is a statistically significant difference between proportions of faults carried by Persistent, Short-lived and Transient classes.

To attempt rejecting $H_{RQ1_0}$, we test whether the proportion of classes in ArgoUML, JFreeChart and XercesJ that compose Short-lived and Transient (respectively Persistent) classes take part (or not) in significantly more faults than those in Persistent (respectively Short-lived and Transient) classes.

Table III
CONTINGENCY TABLE AND FISHER TEST RESULTS IN ARGOUML, JFREECHART AND XERCESJ FOR PERSISTENT, NON-PERSISTENT CLASSES (SHORT-LIVED AND TRANSIENT CLASSES) WITH AT LEAST ONE FAULT

| | Faulty | Clean |
|---|---|---|
| ArgoUML's Non-Persistent classes | 400 | 370 |
| ArgoUML's Persistent classes | 326 | 915 |
| JFreeChart's Non-Persistent classes | 312 | 657 |
| JFreeChart's Persistent classes | 366 | 927 |
| XercesJ's Non-Persistent classes | 268 | 277 |
| XercesJ's Persistent classes | 170 | 508 |
| The Sum of Non-Persistent classes | 980 | 1304 |
| The Sum of Persistent classes | 862 | 2350 |
| Fisher's test | 2.2e-16 | |
| Odd-ratio | 2.048582 | |

*3) Results:* Table III presents a contingency table for ArgoUML, JFreeChart and XercesJ that reports the number of (1) Short-lived and Transient classes that are identified as fault-prone; (2) Short-lived and Transient classes that are identified as clean; (3) Persistent classes that are identified as fault-prone; and, (4) Persistent classes that are identified as clean. The result of Fisher's exact test and odds ratios when testing $H_{RQ1_0}$ are significant. The p-value is less then $0.05$ and the odds ratio for fault-prone Short-lived and Transient classes is two times higher than for fault-prone Persistent classes.

[9] http://sourceforge.net/tracker/index.php?func=detail&aid=814424&group_id=15494&atid=365494

[10] http://sourceforge.net/tracker/index.php?func=detail&aid=1950037&group_id=15494&atid=115494

[11] http://www.r-project.org

We can answer to **RQ1** as follows: we showed that Persistent classes are significantly less fault-prone than Short-lived and Transient classes.

### *B. What is the relation between class co-evolution and fault-proneness?*

*1) Motivation:* The goal of analysing dependencies among co-evolved classes (clusters of classes exhibit similar evolution profiles) is to check if the proportion of faults fixed by maintaining co-evolved classes are significantly more than faults fixed using not co-evolved classes.

*2) Method:* The Chi-Square test is used to test the different proportions of faults fixed for co-evolved and not co-evolved classes. Indeed, the Chi-Square statistic compares the tallies or counts of categorical responses between two (or more) independent groups.

We test for statistical significance to verify the null hypothesis that we state as:

- $H_{RQ2_0}$: There are no statistically significant between proportions of faults involving co-evolved classes or not co-evolved classes in the three programs.

If we reject the null hypothesis $H_{RQ2_0}$, then we explain the rejection either as:

- $H_{RQ2_1}$: The proportion of faults carried by co-evolved classes is not the same as the proportion of faults carried by not co-evolved classes.

To attempt rejecting $H_{RQ2_0}$ we test whether the proportion of co-evolved classes in ArgoUML, JFreeChart and XercesJ take part (or not) in significantly more faults than other classes.

*3) Results:* We use in this test the contingency Table IV, where rows represent the number of faults involving co-evolved classes and the number of faults involving non co-evolved classes. The result of Chi-Square test and odds ratios when testing $H_{RQ2_0}$ are significant. The p-value is less then $0.05$ and we can reject the null hypothesis.

Table IV
CONTINGENCY TABLE AND CHI-SQUARE TEST RESULTS IN ARGOUML, JFREECHART AND XERCESJ FOR FAULTS FIXED BY CO-EVOLVED (CC)OR NOT CO-EVOLVED CLASSES (NCC)

| | Faults involving CC | Faults involving NCC |
|---|---|---|
| ArgoUML | 126 | 92 |
| JFreeChart | 69 | 61 |
| XercesJ | 19 | 15 |
| Chi-Square | 0.01859 | |

We can answer to **RQ2** as follows: faults fixed by maintaining co-evolved classes are significantly more than faults fixed using not co-evolved classes.

## VII. RELEVANCE AND DISCUSSION

### *A. Phylogenetic Tree and Program Evolution*

Phylogenetic tree, is an essential tool in the study of biological evolution since the time of Charles Darwin. In this study, we showed that such trees can be used sufficiently to show the evolution of programs and to spot the design decay phenomenon in software engineering. Indeed, phylogenetic tree shows a representation, simple to understand and quick to generate, of the evolution of a huge amount of data (tens of versions having thousands of classes). Such trees facilitate to developers the comprehension and the detection of design decay by showing major change in the program versions. The phylogenetic tree shows, also, stability periods of programs: in these periods, we pass from one version to another without adding new classes in the program *i.e.*, a new version is published with the same classes, but by fixing some problems such as faults or by adding some functionalities in existing classes.

### *B. Class Lifetime and Fault-proneness*

In this paper, we combine information obtained from class evolutionary history and from bug reports to obtain a clearer picture of the evolution of object-oriented program. This is a key knowledge for a maintenance activity, because it allows us to detect the critical parts of the program that represent the starting point for a maintenance process. We found that Non-Persistent classes should be spotted and well-understood before maintaining the programs as these classes are more fault-prone. Special attention must be given to these entities to keep the design intact during program evolution because the instability of these classes could have a negative impact on the fault-proneness of the program.

### *C. Similarities in Classes Evolution Profiles*

While co-change dependencies analysis reports the sets of classes that are often changed together, our approach reports the sets of classes that evolve in parallel ways and not necessarily at the same time. To the best of our knowledge, previous co-change approaches did not use method such as structure-based and text-based similarities to identify class renamings and, therefore, they could not report co-change or co-evolution relations among renamed classes. In this paper, we noted that such relations describe implicit design dependencies and source code evolution. Thus, special attention must be given to these relations to keep the design intact during program maintenance activity. If numerous co-evolution relations exist, Profilo sort the sets of results depending on the number of static relationships among co-evolved classes in order to help the developers to focus on those that potentially led to a design flaw or to mistakes in maintaining classes together.

### *D. Threats to the Study Validity*

Some threats limit the validity of our empirical study.

**Construct Validity:** Construct validity threats concern the relation between theory and observations. In this study, they could be due to the errors of the implementation. They could also be due to an imprecision of our measurements of

the distance between different couples of class-profile and–or different couples of version-profile. We believe that this threat is mitigated by the facts that we validated Profilo results manually and checked external sources of information (bug reports and others).

**Conclusion validity:** Threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the statistical test that we used, in **RQ1** and in **RQ2**. We cannot claim causation, but our discussion tries to explain why some classes could have been subject to faults.

**Reliability Validity:** Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to re-implement our approach and replicate our empirical study. The programs, change logs, and raw data to obtain our observations are available online at http://www.ptidej.net/downloads/experiments/csmr13a/.

**External Validity:** External Validity concern the generalisation of our findings. Although we performed our analyses on three different programs, belonging to different domains and with different sizes, we are aware that further empirical validations on a larger set of programs would be beneficial to better support our findings. We cannot assert that our results and observations are generalisable to any other program and the fact that all the analysed programs are open source and are developed with Java may reduce this generability.

## VIII. Related Work

Our work relates to different research directions: fault-proneness, program evolution, detection of class evolution, and co-evolution.

### A. Fault-proneness

Nagappan and Ball [21] performed a study on the influence of code churn [22] on the fault density. They found that relative code churn was a better predictor than absolute churn. Moser *et al.* [9] used metrics (*e.g.* code churn, past faults and refactorings, etc.) to predict the presence/absence of faults in files of Eclipse. Hassan and Holt [8] proposed heuristics to analyse fault proneness and they found that recently modified and fixed classes were the most fault-prone. Ostrand *et al.* [23] predict faults on two industrial systems, using change and fault data. Bernstein *et al.* [24] used fault and change information in non-linear prediction models. Zimmermann and Nagappan [25] used dependencies between binaries in Windows server 2003 to predict faults. Marcus *et al.* [26] used a cohesion measurement based on LSI for fault prediction. Neuhaus *et al.* [27] used a variety of features of Mozilla, such as past faults, package imports, call structure, to determine fault vulnerabilities. Previous approach on fault-proneness out there did not link class evolution behaviors to faults. In this paper, we spotted the links between software evolution and fault-proneness.

### B. Program Evolution

The phenomenon of software aging is the result of software evolution. Parnas [28] suggested that programs suffer from various aging problems such as increasing complexity, faults, unstructured code, feature overloading, etc. Eick *et al.* [6] suggested that a code is decayed if it is more difficult to maintain than it used to be. We believe, like the above cited authors, that code decay is essentially the result of program evolution. The design of a program deviates from its planned form with every new version of program to incorporate new features by implementing new classes and–or deleting, refactoring and changing old classes. We relate the evolution of classes in object-oriented programs and fault-proneness to emphasize program evolution consequences. Fraser [16] presented DiffTree to infer a phylogenetic tree from related programs. It described the retrospective computation of version trees for a set of programs, without mining source code control systems.

DiffTree compared set of codes with one another, and presents a parsimonious phylogenetic tree for them. It can also help to identify cases where a repair made to one version was missed in others. We share with the author the idea that is interesting to identify commonalities and divergences among versions to acknowledge the contributions of each version relative to one another. Karim *et al.* [29] described a method for constructing phylogeny models that used $n$-perms to match possibly permuted code and to discover malicious programs, such as viruses and worms, frequently related to previous programs through evolutionary relationships. These last two approaches are the closest to our work. They infer a phylogenetic tree from related programs but do not analyse the classes lifetime and co-evolution relations among them. In addition, our work differs in the aspects considered: they considered differences between versions in term of lines of code and did not consider types of changes or the impact of change in term of classes and program architecture, while we consider differences between versions in term of object-oriented structures (classes) used, modified, refactored, and renamed.

### C. Class Evolution

Many approaches exist to analyse classes in programs based on their relative evolution. Lanza *et al.* [30] presented an evolution matrix to display the evolution of the classes of a program. Each column of the matrix represents a version of the program, while each row represents the different versions of the same class. Lanza *et al.* considered that two classes in two different versions are the same if they have the same name. Then, the authors presented a categorisation of classes based on their visualisation. Our work differs in the level of granularity and on the aspects considered. Indeed, Lanza *et al.* considered only class name to identify classes in different version. Thus, the same class in two different versions are considered two different classes if they are renamed. To

overcome this issue, we use a set of structure-based and text-based similarities to identify class renamings in programs.

UMLDiff [31] compared and detected the differences between the classes of two object-oriented program versions. It extracts the history of the program evolution, in terms of the additions, removals, moves, renamings, and signature-changes of classes. UMLDiff then assigned a stability type to each class: short-lived (they exist only in a few versions of the program), idle (they rarely undergo changes after their introduction in the program), and active (they keep being modified over their whole lifespan). In contrast to UMLDiff, our approach aims to present an understandable model of the evolutionary history of classes by modeling class-profiles with a bit vector model. Other than the simplicity of this model, bit vector allow for significant effectiveness and efficiency on the analysis of the evolution of large programs because it allows small arrays of bits to be stored and manipulated with efficient operations. Further, UMLDiff cannot relate the evolutionary history of classes with fault-proneness.

Kpodjedo *et al.* [32] [18] proposed to identify all classes that do not change in the history of a program, using an Error Correcting Graph Matching algorithm (ECGM). They studied the evolution of programs and recovered traceability links between subsequent class diagrams. Their approach identified evolving classes that maintain stable static relations with other classes and that constitute the stable backbone (tunnel) of programs. In addition to the detection of these stable backbones, our approach groups classes based on their evolution profiles and use these groups to detect co-evolution among them.

Demeyer *et al.* [33] presented an approach to understand how object-oriented programs have evolved by discovering which refactoring operations have been applied from one version of the software to the next. Antoniol *et al.* [34] adopted techniques inspired by Information Retrieval (IR) approaches to automatically identify evolution discontinuities when analyzing the evolution of object-oriented source code at class level. These two last approaches were useful to identify some replacement, merge and refactoring during the evolution of programs but they cannot relate program evolution with faults proneness.

### D. Co-evolution Relationships

The maintenance of a program involves handling a large number of classes. Indeed, classes that exhibit similar evolution profiles, due to interdependencies among them, are considered as co-evolved classes [10]. Xing *et al.* [31] analysed the evolution profile for each class. The class-evolution profile reports the complete history of changes made to an individual class in each subsequent version. Furthermore, they examined clusters of classes that changed in very similar ways for a substantial period of time. However, their approach cannot detect co-evolved classes involved in the same fault. Antoniol *et al.* [35] presented an approach to detect similarities in classes evolution profiles starting from past maintenance. They applied the LPC/Cepstrum technique to identify in version-control systems the classes that evolved in the same or similar ways. Their approach identified co-evolved classes but cannot report the different lifetime and relate these types with fault-proneness. Several approaches identify co-changes among artefacts, *e.g.*, [36], [37], and [13], which represent the (often implicit) dependencies or logical couplings among artefacts that have been observed to frequently change together [38]. Typically, two artefacts are co-changing if they were changed by the same author and with the same log message in a time-window of less than 200 ms [37]. Co-change is one aspect of co-evolution. Indeed, if two classes co-changed then they co-evolved. But if two classes co-evolved then they not necessarily co-changed.

## IX. Conclusion and Future Work

We described a novel approach to analyse programs evolution and to trace fault-proneness. One of goals addressed in this paper is how we can relate the evolution of classes in object-oriented programs with fault-proneness. The concepts of class lifetime, co-evolution, and phylogenetic tree helped us to describe and to identify the reasons that have driven the programs' codes to their current states. We showed that Persistent classes are significantly less fault-prone than other classes and that faults fixed by maintaining co-evolved classes are significantly more than faults fixed using not co-evolved classes. Profilo draw, also, informed conclusions about the relation between maintenance tasks and fault-proneness in order to help developers to understand evolution trends and to maintain the programs correctly.

**Work-in-progress** aims at (1) analysing further co-evolution relations by replicating our study with other larger programs, (2) performing a comprehensive study of the relationships between class lifetime and change-proneness, and (3) identifying the lifetime followed by design motifs such as design patterns and anti-patterns.

## Acknowledgment

This work has been partly funded a FQRNT team grant and the Canada Research Chair in Software Patterns and Patterns of Software. We gratefully thank Massimiliano Di Penta and Giuliano Antoniol for their generous comments.

## References

[1] M. Lehman, J. F. R. an P.D. Wernick, D. Perry, and W. Turski, "Metrics and laws of software evolution-the nineties view," in *Fourth International Software Metrics Symposium*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 20–.

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

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

[4] S. Hassaine, Yann-Gaël, S. Hamel, and A. Giuliano, "Advise: Architectural decay in software evolution," in *Proc. 16th European Conference on Software Maintenance and Reengineering*. ACM, 2012, pp. 267–276.

[5] 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.*, pp. 277–306, 2005.

[6] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Trans. Softw. Eng.*, pp. 1–12, 2001.

[7] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 86–96.

[8] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. IEEE Computer Society, 2005, pp. 263–272.

[9] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 181–190.

[10] Z. Xing and E. Stroulia, "Data-mining in support of detecting class co-evolution," in *Proc. 16th Internatioanl Conferance on Software Engineering and Knowledge Engineering*. Citeseer, 2004.

[11] B. Hall, *Phylogenetic trees made easy: a how-to manual*. Sinauer Associates Sunderland, Massachusetts, 2004.

[12] W. Fitch, E. Margoliash *et al.*, "Construction of phylogenetic trees," *Science*, vol. 155, no. 760, pp. 279–284, 1967.

[13] F. Jaafar, Y. Guéhéneuc, S. Hamel, and G. Antoniol, "An exploratory study of macro co-changes," in *Working Conference on Reverse Engineering (WCRE)*. IEEE, 2011, pp. 325–334.

[14] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multi-layered framework for design pattern identification," *Transactions on Software Engineering (TSE)*, pp. 667–684, 2008.

[15] 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, 2004, pp. 301–314.

[16] C. Fraser, "Difftree: Inferring phylogenies for evolving software," Microsoft Research, Tech. Rep., 2005.

[17] R. Real and J. Vargas, "The probabilistic basis of jaccard's index of similarity," *Systematic Biology*, pp. 380–385, 1996.

[18] S. Kpodjedo, F. Ricca, P. Galinier, Y.-G. Guéhéneuc, and G. Antoniol, "Design evolution metrics for defect prediction in object oriented systems," *Empirical Softw. Engg.*, 2011.

[19] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Trans. Software Eng.*, pp. 728–738, 1984.

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

[21] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. ACM, 2005, pp. 284–292.

[22] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[23] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *Software Engineering, IEEE Transactions on*, vol. 31, no. 4, pp. 340 – 355, 2005.

[24] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, ser. IWPSE '07. ACM, 2007, pp. 11–18.

[25] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. ACM, 2008, pp. 531–540.

[26] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 287–300, 2008.

[27] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. ACM, 2007, pp. 529–540.

[28] D. L. Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society Press, 1994, pp. 279–287.

[29] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, pp. 13–23, 2005.

[30] M. Lanza and S. Ducasse, "Understanding software evolution using a combination of software visualization and software metrics," Citeseer, 2002.

[31] Z. Xing and E. Stroulia, "Analyzing the evolutionary history of the logical design of object-oriented software," in *IEEE Transactions on Software Engineering*, vol. 31. IEEE Computer Society, 2005, pp. 850–868.

[32] 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 *European Conference on Software Maintenance and Reengineering*, 2009, pp. 179–188.

[33] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '00. New York, NY, USA: ACM, 2000, pp. 166–177.

[34] G. Antoniol, M. D. Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *In IWPSE*. IEEE Computer Society, 2004, pp. 31–40.

[35] G. Antoniol, V. F. Rollo, and G. Venturi, "Linear predictive coding and cepstrum coefficients for mining time variant information from software repositories," in *Proceedings of the international workshop on Mining software repositories*. ACM Press, 2005, pp. 1–5.

[36] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Software Eng.*, pp. 574–586, 2004.

[37] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 563–572.

[38] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 1998, p. 190.