Analyzing Software Evolution and Quality by Extracting Asynchrony Change patterns

Fehmi Jaafar, Angela Lozano, Yann-Gaël Guéhéneuc, Kim Mens

PII: S0164-1212(17)30094-8
DOI: 10.1016/j.jss.2017.05.047
Reference: JSS 9960

To appear in: *The Journal of Systems & Software*

Received date: 27 May 2016
Revised date: 7 February 2017
Accepted date: 12 May 2017

Please cite this article as: Fehmi Jaafar, Angela Lozano, Yann-Gaël Guéhéneuc, Kim Mens, Analyzing Software Evolution and Quality by Extracting Asynchrony Change patterns, *The Journal of Systems & Software* (2017), doi: 10.1016/j.jss.2017.05.047

Highlights

- An empirical study to identify the more risky part of code in software systems.
- Cloned files that follow Asynchrony change patterns are more fault-prone.
- Anti-patterns following Asynchrony change pattern are up to five times faultier.

Analyzing Software Evolution and Quality by Extracting Asynchrony Change patterns

Fehmi Jaafar[1], Angela Lozano[b], Yann-Gaël Guéhéneuc[c], Kim Mens[1]

[a]*Polytechnique Montreal, Canada*
[b]*The Software Languages Lab. Vrije Universiteit Brussel, Belgium*
[c]*Ptidej Lab. Polytechnique Montreal, Canada*
[d]*The RELEASeD group, Université catholique de Louvain, Belgium*

Abstract

Change patterns describe two or more files were often changed together during the development or the maintenance of software systems. Several studies have been presented to detect change patterns and to analyze their types and their impact on software quality. In this context, we introduced the Asynchrony change pattern to describes a set of files that always change together in the same change periods, regardless developers who maintained them. In this paper, we investigate the impact of Asynchrony change pattern on design and code smells such as anti-patterns and code clones.

Concretely, we conduct an empirical study by detecting Asynchrony change patterns, anti-patterns and code clones occurrences on 22 versions of four software systems and analysing their fault-proneness. Results show that cloned files that follow the same Asynchrony change patterns have significantly increased fault-proneness with respect to other clones, and that anti-patterns following the same Asynchrony change pattern can be up to five times more risky in terms of fault-proneness as compared to other anti-patterns. Asynchrony change patterns thus seem to be strong indicators of fault-proneness for clones and anti-patterns.

Keywords: Change Patterns, Anti-patterns, Clones, Fault-proneness, Software Quality

1. Introduction

A major concern of practitioners and researchers in software engineering is extracting knowledge from software repositories. Such repositories enclose

a large amount of data on files and changes applied on these files. This data includes the source code, comments about bugs and failures, modifications made to the different source code files, times of changes, developers' identities, *etc.*

Such software repositories present a rich resource of data to discover models of software evolution and to quantify or predict the quality of a software system. More specifically, they contain relevant data to spot co-changing files, or to discover files that present a lower quality. Indeed, using data extracted from software repositories, we can detect anti-patterns, *i.e.*, patterns that indicate weaknesses in software design and may affect the software quality and maintainability [1], as well as clones, *i.e.*, duplicated code that may specify a poor or lazy programming style and may increase the risk of unintentional incomplete changes and bugs [2]. In addition, we can detetct change patterns, *i.e.*, that highlight co-changing group of files [3]. Our previous studies analyzed independently the impact of anti-patterns [4, 5], clones [6], and change patterns [7] on fault-proneness. In this study, we build on these papers and we continue to investigate the impact of such patterns on software evolution and quality.

Previous work [4, 5, 8, 9] has shown that such patterns could be related to each other. In addition, we have described [7] the Asynchrony change pattern to define co-change relations between related files in the source code. We reported that the detection and analysis of such relations may allow us to detect critical fragments of software systems that represent sets of more risky files in terms of fault-proneness. Special attention must thus be given to these co-change relations to keep the design in good shape.

Therefore, the aim of this paper is threefold: first, we investigate whether co-occurrences of anti-patterns and Asynchrony change pattern may increase the fault-proneness of files. Second, we investigate whether clones following Asynchrony change pattern are more fault-prone than others. Third, we analyze the intersection between anti-patterns and clones.

Using occurrences of Asynchrony change patterns, anti-patterns, and clones extracted from two open-source systems, we answer the following research questions:

- **RQ1:** *What is the impact on fault-proneness when a file participates both in anti-pattern occurrences and Asynchrony change pattern occurrences?*

- **RQ2:** *What is the impact on fault-proneness when a file participates both in clones and Asynchrony change pattern occurrences?*
- **RQ3:** *What is the fault-proneness of files that participate in anti-patterns and clones?*

We observed that, in four open source systems analysed in this study, files participated in Asynchrony change patterns are more fault-prone if they belong to clones or anti-pattern occurrences. In addition, we found that classes presenting co-occurrence of anti-patterns and clones are at least six times more risky to occur faults than the rest of classes.

The remainder of this paper is organized as follows. Section 3 presents our methodology. Section 4 describes the empirical study. Section 5 presents the results of the study, while Section 6 discuss the threats to validity. Section 2 relates our study with previous work. Finally, Section 7 concludes the paper and outlines several avenues for future work.

2. Related Work

During the past years, different approaches have been developed to analyze the evolution of the quality of software systems, in particular, specifying change patterns, detecting anti-patterns, spotting code clones, and analyzing their impact on fault proneness. This section discusses some of the literature that aims at investigating these problems.

2.1. Change patterns

Ying *et al.* [10] and Zimmermann *et al.* [11] applied Association Rules to identify co-changing files. An association-rule algorithm extracts frequently co-changing files of a commit into sets that are regarded as change patterns to guide future changes. Such an algorithm uses co-change history in a version control system and avoids a source code dependent parsing process.

Ceccarelli *et al.* [12] and Canfora *et al.* [13] proposed the use of a vector-based auto-regression model, a generalisation of univariate auto-regression models, to capture the evolution and the inter-dependencies between multiple time series representing changes to files. They used the bivariate Granger causality test to identify if the changes to some files are useful for forecasting changes to other files. They concluded that the Granger test is a viable approach to change impact analysis and that it complements existing approaches like Association Rules to capture co-changes.

Bouktif *et al.* [3] defined the general concept of change patterns and described one such pattern, Synchrony, that highlights co-changing groups of artefacts. Their approach used a Sliding Window algorithm as in [11] to build Synchrony change pattern occurrences.

Discussion:

In our previous work [7], we showed that several approaches could find files having very similar maintenance evolution history, but they could not detect co-changed files maintained by different developers or in separated periods of time, which could lead to missed co-changing files and change propagation scenarios. Indeed, using Macocha, we are able to detect the Asynchrony change pattern occurrences that provide an in-depth information about co-changing files to developers that help in explaining bugs, managing development teams, and performing traceability analysis. In this paper, we study the impact of being of part of an the Asynchrony change pattern occurrence for clones and anti-patterns.

2.2. Anti-patterns Definition and Detection

Code and design smells are poor solutions to recurring implementation and design problems. Indeed, several previous works [14][15] have been written to specify them. Actually, code smells are indicators or symptoms of the possible presence of design smells. In the following, we will present a set a papers that relate the detection of code and design smells with the evaluation of the quality of software systems.

Webster [14] reported that an anti-pattern describes a frequently used solution to a problem that generates ineffective or decidedly negative consequences. Brown *et al.* [16] presented 40 anti-patterns, which are often described in terms of lower-level code smells. These books provide in-depth views on heuristics, code smells, and anti-patterns aimed at a wide academic and practitioners audience. They are the basis of all the approaches to detect anti-patterns.

Van Emden *et al.* [17] developed the JCosmo tool, which parses source code into an abstract model (similar to the Famix meta-model). JCosmo uses primitive and rules to detect the presence of smells and anti-patterns. It could visualize the code layout and display anti-patterns locations to help developers assess code quality and perform refactorings.

Marinescu *et al.* developed a set of detection strategies to detect anti-patterns based on metrics [18]. They defined history measurements which summarize the evolution of the suspects parts of code. Then, they showed

that the detection of God Classes and Data Classes can become more accurate using using historical information of the suspected flawed structures.

Discussion:

We share with all the above authors the idea that anti-patterns detection is a powerful mechanism to assess code quality, in particular to study whether the existence of anti-patterns and their evolutions make the code more difficult to maintain. In this paper we perform a study to analyse co-change dependencies across anti-patterns to understand their impact on fault-proneness. In our previous study [5], we reported the results of an empirical study, performed on three object-oriented systems, which provides empirical evidence of the negative impact of dependencies with anti-patterns on fault-proneness. We found that having static relationships with anti-patterns (such as use, creation, association, aggregation, and composition relationships) can significantly increase fault-proneness. Indeed, we identified all the instances of the anti-patterns and design patterns of interest in the different analyzed systems. Then, we detect the static relationships of the classes in these instances with the rest of the classes. Thus, we assume that an anti-pattern A has a static relationships with a class C if at least one class belonging to A has a use, association, aggregation, or composition relationships with C. We also found that classes having co-change dependencies with anti-patterns are more fault prone than others. In this paper, we continue to investigate the impact of specific source code motifs, such as anti-patterns and change patterns, and we report new results that confirm, within the limits of the threats to its validity, the conjecture in the literature that anti-patterns have a negative impact on software system quality, especially when such anti-patterns follow the same Asynchrony change pattern occurrence.

2.3. Software Clones Definition and Detection

As we defined in several previous works, a clone is a source code fragment whose structure is identical or very similar to the structure of another code fragment. Cloned code is a consequence of a frequent programming practice: copying a piece of functionality and pasting it in another context where it is adapted. Baker [19] reported that software systems contain about 30% of redundant code, often referred as code clones. Ducasse *et al.* [20] observed that clones exist at rates of over 50% of the effective lines of code ELOC in a particular COBOL system. Related works reported different clone rates in software systems. For example, Koschke and Bazrafshan [21] found that

the analysed open source applications have cloning rates of 1-12%. Cheung *et al.* [22] reported that JavaScript web applications have 95% of inter-file clones and 9197% of widely scattered clones. Li *et al.* [23] found that PHP web applications have cloning rates of 5-82%.

Several works indicated that code cloning is commonly a bad smell [24]. However, other works argue that, in some contexts, clones can be a practical way to actually design or implement a system. For example, Kapser and Godfrey [25] introduced eight cloning patterns that they had uncovered during projects post-mortem. These patterns present numerous motivations for cloning, such as Templating, which occurs when the desired behavior is already known and an existing solution closely satisfies this need.

Numerous tools have been developed for clone detection. In the following, we presented some preeminent related work. Ducasse *et al.* [26] developed a language independent clone detection tool *duploc*. The author defined a line based comparison by using a dynamic pattern matching. Indeed, their approach allows spotting all occurrences of a pattern in a text.

The Simian[1] tool is able to detect clones in different programming languages. If this tool does not identify programming language of the source file, then it treats it as a natural text file to find clones. Lee and Jeong presented the *Similar Data Detection* tool [27] to analyze clones in large systems. Their tool is based on generating index and inverted index for code fragments and their positions. Then, Lee and Jeong used an n-neighbor distance algorithm to locate clones. Cordy *et al.* [28] detected near-miss clones in HTML web pages using island grammar to identify and extract all structural fragments. This grammar represents the bulk of the code (HTML text in our experiment) as sequences of uninteresting tokens known as water. Then, potential clones are compared to each other using the Unix diff command *i.e.,* a standard utility on most Unix systems that can be used to determine the differences between two files (usually on a line-by-line basis), and display them in a number of formats.

Metrics extracted from source code are compared to assess similarity [29][30]. These metrics are calculated from names, layout, expressions and control flow of functions and used with abstract syntax tree to detect clones. *Koschke et al.* [31] presented a clone detection tool using a parser to generate abstract syntax tree AST. Then the AST is serialized and input

[1] `http://www.harukizaemon.com/simian/`

to suffix tree. The technique is able to detect syntactic units which are not possible by applying suffix tree only. In fact, abstract syntax trees and parse trees are commonly used models when source code is to be transformed into tree structures [32] [33] [34].

Komondoor and Horwitz [35] and krinke [36] presented a clone detection tools based on a program dependency graph (PDG). A program dependency graph represents control and data flow dependencies of a function of source code. Graph representation of source program is partitioned depending upon the behavior of source code fragment. The partitioning algorithm and substring comparison are used to detect similarity. However, it has shown in an evaluation experiment reported by Bellon *et al.* [37], that PDG based clone detection approaches can be slow in detection of contiguous clones. Thus, Higo *et al.* proposed a technique for improving the PDG-based clone detection by introducing new links between every pair of two nodes whose program elements are consecutive in the source code. Authors reported that the proposed technique improved detection capability of the PDG-based technique.

CCFinder [38], a token based clone detection tool, uses suffix tree matching algorithm to find clones. Indeed, in token based clone detection techniques, firstly, tokens are extracted from the source code by lexical analysis. Then some set of tokens (at a specific granularity level) is formed into a sequence. Suffix tree or suffix array based token by token comparison is the heart of token based clone detection algorithms. CCFinder is popular among researchers and has been widely used for code clone analysis, code clone management, etc. In addition, CCFinder shows the highest recall and reasonable precision in comparison with other tools [37].

Discussion:

Geiger et al. [8] have shown that a high number of files shared clones and co-changes. However, the results of a statical test on the formula proposed to express co-change relations in function of cloning relations were not statistically significant. In our previous work [6], we showed that the relation between cloning and co-changes requires a relaxed definition of changes, balanced by a stricter definition of co-changes. Given that clone groups may change in the same way but not necessarily at the same time, the Asynchrony change pattern might be more appropriate to analyze consistent changes in clones than commits. Although previous attempts at providing evidence that cloned code requires to be co-changed have been unsuccessful, we showed in [6] that cloned files tend to follow the Asynchrony change pattern with files with which they share cloned fragments. Moreover, files requiring consistent

changes (those that co-change with their clones) are indeed likely to create bugs. In this paper, we present, to the best of our knowledge, the first empirical evidence of the impact of cloned files that follow the same change pattern on fault proneness.

2.4. Fault-proneness

The most studied and traditional approach for fault prediction is to relate software faults to the size and complexity of code [39] and [40]. Chidamber and Kemerer [41] proposed a suite of object-oriented design metrics which has been substantiated by several theoretical and empirical studies [42] and [43]. Results show that the more complex the code is, the more faults exist in it. In addition, the size of source code is one of the best indicators for fault proneness. Hassan and Holt [44] proposed heuristics to analyze fault proneness. They found that recently modified and fixed classes were the most fault-prone.

D'Ambros *et al.* [45] reported that there was a correlation between change coupling and defects which is higher than the one observed with complexity metrics. Further, defects with a high severity seem to exhibit a correlation with change coupling which, in some instances, is higher than the change rate of the components. They also enriched bug prediction models based on complexity metrics with change coupling information. Marcus *et al.* [46] used a cohesion measurement based on Latent semantic indexing (LSI), an indexing and retrieval method to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of source code. LSI is based on the principle that words that are used in the same contexts tend to have similar meanings. The authors used LSI for fault prediction and they stated that structural and semantic cohesion directly impacts the understandability and readability of the source code.

Ostrand *et al.* [47], Bernstein *et al.* [48], and Neuhaus *et al.* [49] predict faults in systems, using change and fault data, while Moser *et al.* [50] used metrics (*e.g.* code churn, past faults and refactorings, etc.) to predict the presence/absence of faults in files of Eclipse.

Khomh *et al.* [1] investigated the impact of anti-patterns on classes in object-oriented systems by studying the relation between the presence of anti-patterns and the change- and fault-proneness of the classes. The authors showed that in almost all releases of the four systems, classes participating in anti-patterns are more change and fault-prone than others.

Several papers explored the relation between clones and bugs. In particular, Aversano et al. found that several late propagations were linked to bug-fixes [51]. Juergens et al. showed that only half of the unintentionally inconsistent changes on clones were related to faults [2]. Indeed, if a code fragment contains a bug and that fragment is cloned at different places, the same bug will be present in all the code fragments. Thus, code cloning can increase the probability of bug propagation [51] [2].

Discussion:

Previous work raised the consciousness of researchers and developers towards the impact of anti-patterns and clones on software quality and maintenance tasks. In our previous work [7], we have showed that if two files are following the same Asynchrony change pattern, thus this implies the existence of (hidden) dependencies between these two files. If these dependencies are not properly maintained, they can introduce faults in a system. In this paper, we aim to understand if the negative effects of specific source code motifs, such as anti-patterns and clones, can increase the fault-proneness of files in software systems, when such motifs follow the same Asynchrony change patterns.

3. Background and Methodology

In this section, we define the different phases of our methodology by listing the used concepts and their background. First of all, as shown in Figure 1, we perform the identification of anti-patterns using the DECOR approach [52]. Second, we use CCFinder [38] to detect duplicated code describing clones in the software systems. Third, we use Macocha [7] to detect Asynchrony change pattern occurrences by mining version-control systems (such as CVS and SVN). Finally, we apply the heuristics by Sliwersky *et al.* [53] to identify fault fix locations and investigate the fault proneness of files participating in anti-patters or clones, and following Asynchrony change patterns.

3.1. Anti-pattern Identification

We use the DECOR approach[52] to specify and detect anti-patterns in each release of the analyzed systems. DECOR is based on a thorough domain analysis of anti-patterns specified in the literature, presents a domain-specific language to model smells, and a tool to spot their occurrences. DECOR uses the Pattern and Abstract-level Description Language PADL [54] meta-model

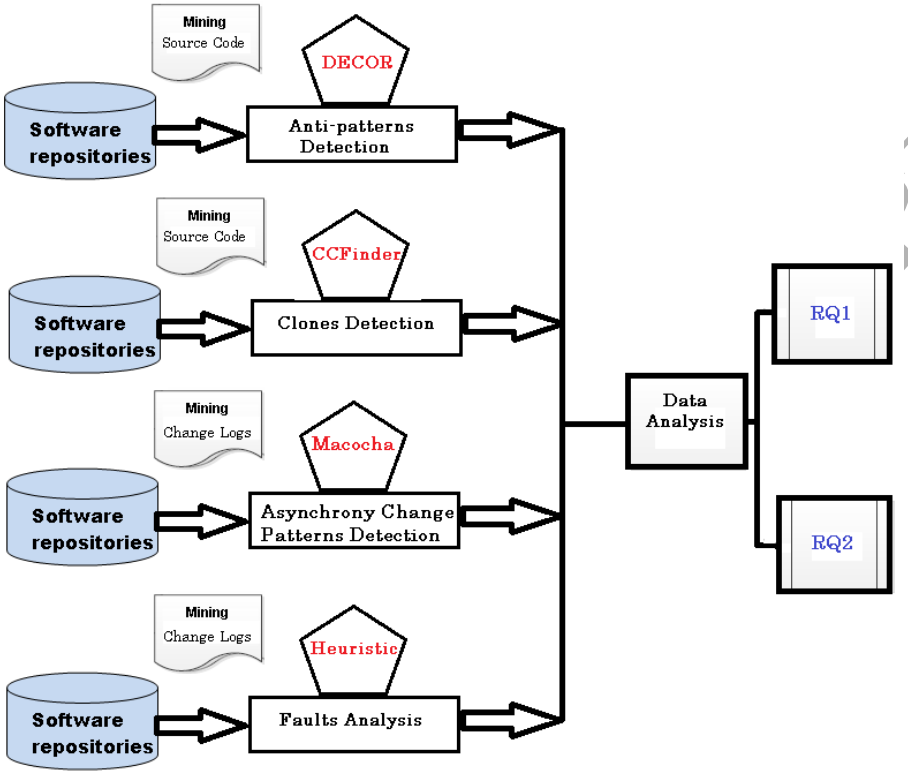

Figure 1: Overview of our approach to study the fault-proneness of clones and anti-patterns that follow Asynchrony change patterns.

and POM framework (Primitives, Operators, Metrics) [55]. PADL is a meta-model used to define the architecture of object-oriented systems [54]. POM is a PADL-based framework that calculates more than 60 metrics in relation with software quality.

We focus on 10 anti-patterns from Brown *et al.* [16]. We choose these anti-patterns because they are representative of problems with data, complexity, size, and the features provided by files [1]. We also use these anti-patterns because they have been used and analyzed in previous work [1] to validate our results by comparing them with previous data [52]. The definitions and specifications of the anti-patterns are beyond the scope of this paper and are available in [56]. The list of anti-patterns considered in this study is as follows:

- AntiSingleton: which provides mutable variables that could be used as

global variables.

- Blob: which is too large and not cohesive enough, that monopolizes most of the processing, takes most of the decisions, and is associated to data classes.
- ClassDataShouldBePrivate: which exposes its fields, thus violating the principle of encapsulation.
- ComplexClass: which has (at least) one large method, in terms of cyclomatic complexity and LOCs.
- LongMethod: which has a method that is overly long, in term of LOCs.
- LongParameterList: which has (at least) one method with a too long list of parameters with respect to the average number of parameters per methods in the system.
- MessageChain: which uses a long chain of method invocations to realize (at least) one of its functionality.
- RefusedParentBequest: which redefines inherited methods using empty bodies thus breaking polymorphism.
- SpeculativeGenerality: which is defined as abstract but that has very few children.
- SwissArmyKnife: which contains methods that can be divided in disjunct set of many methods, thus providing many different unrelated functionalities.

Based on an extensive empirical study performed on nine software systems [52], DECOR approach has a recall of 100% and a precision greater than 50%, with an average precision greater than 60%.

3.2. Clone Identification

We use CCFinder [38] to detect clones in our empirical study. The author of this tool defines a clone relation as an equivalence relation (*i.e.,* reflexive, transitive, and symmetric relation) on code fragments. A clone relation describe two fragments of code with the same token sequences. Indeed, CCFinder uses a token-based clone process in which the input is source files

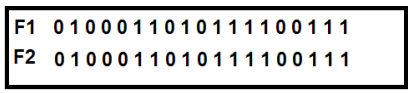

Figure 2: Files `F1` and `F2` follow the Asynchrony change pattern

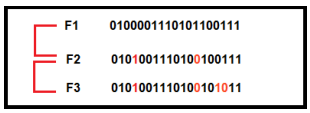

Figure 3: Three different bit vectors showing approximate Asynchrony change pattern

and the output is clone pairs. First, each line of source files is divided into tokens in accordance with the programming language lexical rules. Then, equivalent pairs of transformed token sequences are detected as clone pairs [38].

We choose CCFinder because, as compared to other clone detection tools, CCFinder has a good performance, scalability, and recall, through a lower precision [57] [58]. Actually, choosing CCFinder can amplify the identification of fragmented clones, *i.e.* clones that are broken up by a few lines of non-cloned fragments of code. For example, Burd and Bailey [57] conducted an experiment to compare three clone detection tools, CCFinder, CloneDR and Covet and two plagiarism detectors JPlag and Moss. The results of their experiment showed that CCFinder had the highest recall (72%) and reasonable precision (72%).

Furthermore, CCFinder does not depend on the syntax of the language, as opposed to abstract syntax tree based detection tools [59]. In this study, clones were identified using the configuration of CCFinder version 10.2.7.

3.3. Change-Pattern Identification

We use Macocha [7] to identify occurrences of the Asynchrony change pattern in software systems.

An Asynchrony change pattern describes two or more changed files that change together during the period of time considered in the study.

To detect occurrences of Asynchrony change pattern, we have to detect the change periods in a software system by identifying all the changes performed closely in time, whatever the developers who committed them and the messages log.

Second, we define a profile as a bit vector that describes whether a file is changed, or not, during each of the change periods of a software system.

The length n of this bit vector is the number of change periods discovered by the KNN algorithm. Indeed, we used the KNN algorithm, a non-parametric learning algorithm, to identify changes occurring close to one another, that is, belonging to a same change period. We showed that, using this algorithm to group changes into change periods, allowed us to improve precision and recall over the state-of-the-art previous approaches [7]. Precision and recall are metrics used in information retrieval studies to measure the accuracy of a detection method. In several previous study [7], [11], and [12], these metrics were defined in terms of the set of retrieved co-changed files produced by a query and the set of relevant co-changed files. Precision is the fraction of co-changed files that are relevant. Recall is the fraction of the co-changed files that are relevant to the query that are successfully retrieved.

Third, similar profiles grouped together represent occurrences of the Asynchrony change patterns, as illustrated in Figure 2. Indeed, Macocha returns, among other results, the following sets of occurrences of change patterns:

- S_{MC}, the set of files with identical profiles in a software system;
- S_{MCH}, the set of files with similar profiles in a software system by using the Hamming distance with $D_H \in]0, 3[$. We choose this threshold in accordance with the results of a previous empirical study [7]. In fact, After analyzing several values of threshold between two profiles in different programs, we found that $D_H \in]0, 3[$ is the best trade-off between precision and recall.

Macocha approach for Asynchrony change patterns identification ensures in average 96% of recall and has a precision greater than 85%.

3.4. Fault Proneness Computation

Finally, we estimate the fault-proneness of a file by relating fault reports (extracted from Bugzilla) and commits to the file. We mine version-control systems to detect changes committed to each file in order to fix faults. A commit contains several attributes such as the dates of the change and the name of the developer who committed the changes. Fault fixing changes are documented in textual reports that describe different types of problems in a software system. These textual reports are stored in issue tracking systems like Bugzilla or Jira. We use a Perl script to parse the SVN/CVS change logs of the subject systems to identify fault fixing commits. This Perl script,

which was used successfully in previous work [1], implements the heuristics by Sliwersky *et al.* [53]. In fact, the authors reported that their heuristics have a precision value of more than 90%. Then, from each fault fixing commit, we extract the list of files that were changed to fix the fault. Indeed, we parse bug reports and commit log messages using bug IDs and specific keywords, such as fault or bug to identify fault-related commits from which we extract the list of files that were changed to fix the faults.

4. Setup of The Study

This section describes the setup of our empirical study. In particular, we present more details about the approach and the analyzed subject systems. Then, we elaborate our analysis method.

The design of our study follows the Goal-Question-Metric (GQM) approach [60]. The *goal* of our study is to investigate the fault-proneness of anti-patterns and clones when they also match the Asynchrony change pattern. The *quality focus* is the analysis of the evolution of software systems and the impact of anti-pattern occurrences and clones on their qualities. The *perspective* is that the results may characterize a primary knowledge for both researchers and practitioners who want to spot specific occurrences of change patterns in order to build future maintenance activities. The *context* of this study is four open source java software systems, namely ArgoUML, JFreeChart, Lucene and XalanJ.

4.1. Analyzed Systems

We use several criteria to select our datasets. First, we select open-source systems, so that other researchers can replicate our experiment. Second, we choose different sizes of systems that represent systems handled by most developers. Third, we selected systems with datasets containing a log history and fault specification information. The selected systems, ArgoUML, JFreeChart, Lucene, and XalanJ span several years and versions and have between hundreds and thousands of files. Table 1 summarizes the data collected showing the case studies, programming language, revisions analyzed, number of Asynchrony change pattern occurrences found in those revisions, the number of files in that time interval, and the number of different files sharing cloned fragments or involving in anti-patterns in that time interval.

ArgoUML is a UML diagramming system written in Java and released under the open-source BSD License. For our study, we extracted a total

Table 1: Descriptive statistics of the object systems

Software systems	ArgoUML	JFreeChart	Lucene	XalanJ
# of files	3,325	1,615	750	1,067
# of snapshots	4,480	2,010	1,125	832
# of releases	11	9	1	1
SLOC average	240,762	181,895	127,699	366,425
# of AntiSingleton	3	38	17	1
# of Blob	100	49	24	34
# of ClassDataShouldBePrivate	51	3	2	18
# of ComplexClass	158	52	27	53
# of LongMethod	336	75	25	110
# of LongParameterList	281	76	38	94
# of MessageChains	162	59	29	54
# of RefusedParentBequest	123	5	3	41
# of SpeculativeGenerality	22	3	1	7
# of SwissArmyKnife	13	26	13	5
# of Clones	847	692	234	766
# of Asynchrony change pattern occurrences	144	425	125	283

number of 4,480 snapshots between September 27th, 2008 and December 15th, 2011.

JFreeChart is a Java open-source framework to create charts. For our study, we considered an interval of observation ranging from June 15th, 2007 (release 1.0.6) to November 20th, 2009 (release 1.0.13α) in which we extracted 2,010 snapshots.

Lucene is an open-source system hosted by Apache and written in Java. It aims to produce high-performance full-text indexing and search software and is used by many popular websites and Java applications. We analysed the development period of the version 1.4 which includes 750 Java files.

XalanJ is a Java open source software library from the Apache Software Foundation that implements and maintains libraries and programs that transform XML documents using XSLT. We analysed the development period of the version 2.6.0 which contains 1,067 Java files.

We choose these periods as we can identify enough occurrences of clones, anti-patterns and Asynchrony change patterns to conduct statistical tests. In addition, from our previous analysis [7], we know that for these periods, we can use the adequate threshold value, without performing any other preliminary analysis, to identify Asynchrony change pattern occurrences while maintaining the best trade-off between precision and recall. Last but not least, as we analysed the same periods in previous work, we were able to

recover and validate the co-change dependencies manually by using previous results.

4.2. Analysis Method

The analysis reported in Section 5 have been performed using the R statistical environment[2]. We use Fisher's exact test [61] because it is a significance test that is considered to be more appropriate for sparse and skewed samples of data than other statistical tests, such as the log likelihood ratio or Pearson's Chi-Squared test [62]. Indeed, this test it is useful for categorical data that result from classifying objects in two different groups. In our study, we examine the significance of the relation between the occurrence of Asynchrony change pattern, anti-patterns or clones, and the number of fault.

Then, we use Chi-squared's test [61] to check whether the difference in fault-proneness between classes sharing the considered anti-patterns and clones and other classes is significant. Chi-squared's test is a nonparametric statistical test used to examine the significance of the association (contingency) between two classifications. Indeed, an inferential test such the Chi-squared's test is used in order to understand whether the overall population medians are significantly different, based on a sample median. In this case, the considered population is the set of classes of the considered systems. The considered sample used in the Chi-squared's test is the set of classes of the specific analyzed version.

We also compute the odds ratio [61] that indicates the likelihood for an event to occur. The odds ratio shows the strength of the association between a predictor and the response of interest. Its advantage is that it can be used to directly compare findings of different study designs.

The odds ratio is defined as the ratio of the odds p of an event occurring in one sample, *i.e.,* the odds that anti-patterns following Asynchrony change pattern are identified as fault-prone to the odds q of the same event occurring in the other sample, *i.e.*, the odds that the rest of files are identified as fault-prone. Thus, if the probabilities of the event in each of the groups are p (faulty files for example) and q (not faulty files), then the odds ratio is: $OR = \frac{p/(1-p)}{q/(1-q)}$. An odds ratio greater than 1 indicates that the event is more likely in the first sample, while an odds ratio less than 1 indicates that it is more likely in the second sample.

[2] `http://www.r-project.org`

Last but not least, we report the effect size that measures the strength of the relationship between two variables on a numeric scale. Indeed, this statistical concept helps us in determining if the difference between in fault-proneness between classes sharing the considered anti-patterns and clones and other classes is real or if it is due to a change of factors.

5. Study Results

Tables 1, 2, 3, and 4 summarizes the data obtained by analyzing ArgoUML and JFreeChart. In this section, we will present in depth the motivation behind each research question, the used method to answer it, and the results collected on its context. We validated the co-change set of files of anti-pattern occurrences and clones manually. Yet, all these sets were small enough so that we were able to recover and validate their co-change dependencies manually in a previous work. We also checked external sources of information provided by bug reports to confirm the results.

5.1. What is the impact on fault-proneness when a file participates both in anti-pattern occurrences and Asynchrony change pattern occurrences?

5.1.1. Motivation

There exist a number of approaches that detect and analyze anti-patterns in source code to alert developers of their occurrence and impact on the quality of software systems [52, 18, 63]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. In parallel, historical information has been used in the context of quality analysis for the purpose of assessing to what extent anti-patterns remained in the system for a substantial amount of time [1, 5]. However, to the best of our knowledge, the use of historical information and software repertoires for identifying the fault-proneness of files participating in anti-patterns while they follow change patterns remains a premiere of this paper. Previous work [1, 64] has shown that anti-patterns have different "bad" properties that make them prone to faults or hard to understand/maintain. Analyzing and calculating the fault-proneness of anti-patterns that follow an Asynchrony change pattern allow programmers to determine vulnerable parts in the source code so that they can prioritize maintenance activities towards the most risky parts of software systems.

5.1.2. Method

First, we identify anti-pattern occurrences in the analyzed systems using DECOR. Second, we identify occurrences of the Asynchrony change pattern using Macocha. Then, we provide quantitative data on the anti-patterns that also match an Asynchrony change pattern in the studied systems. Finally, we perform a statistical analysis by verifying the null hypothesis which we state as follows:

- H_{RQ1_0}: There is no statistically significant difference in fault-proneness between files both affected by anti-patterns and showing the Asynchrony change pattern and those files affected by anti-patterns but not showing the Asynchrony change pattern.

5.1.3. Results

Table 2: Contingency table and Fisher test results for ArgoUML, JFreeChart, Lucene, and XalanJ for anti-patterns that match the Asynchrony change pattern (AACPA: Anti-patterns that follow Asynchrony change patterns with other anti-patterns, AACPNA: Anti-patterns that follow Asynchrony change patterns with other files)

	Faulty	Non-faulty
(1) Number of AACPA in ArgoUML	132	62
(2) Number of AACPNA in ArgoUML	164	136
Fisher's test for ArgoUML	0.001969	
Odds-ratio for ArgoUML	1.76354	
(1) Number of AACPA in JFreeChart	38	145
(2) Number of AACPNA in JFreeChart	26	399
Fisher's test for JFreeChart	2.49e-07	
Odds-ratio for JFreeChart	4.010807	
(1) Number of AACPA in Lucene	19	78
(2) Number of AACPNA in Lucene	13	200
Fisher's test for Lucene	0.0004933	
Odds-ratio for Lucene	3.729169	
(1) Number of AACPA in XalanJ	51	21
(2) Number of AACPNA in XalanJ	55	45
Fisher's test for XalanJ	0.02517	
Odds-ratio for XalanJ	4.010807	
Effect size for XalanJ	1.97911	

Fault-proneness refers to whether an artefact underwent at least one fault fix in the system life cycle [1]. In our study, we report the cases when faults, anti-patterns, and Asynchrony change patterns co-occur at the same period

of time. Moreover, we are analysing the scenarios when the fault fix occur during the period when a file is involved in an anti-pattern and showing Asynchrony change pattern.We noticed here that the Asynchrony change pattern can involve two or more files. A file is considered as AACPNA when it belongs to anti-pattern and it is involved in Asynchrony change pattern with other files. Table 2 presents a contingency table for ArgoUML, JFreeChart, Lucene, and XalanJ that reports the number of anti-patterns that share Asynchrony change patterns with other anti-patterns and identified as faulty; anti-patterns that share Asynchrony change patterns with anti-patterns and identified as non-faulty; other anti-patterns identified as faulty and participating in Asynchrony change patterns with other files; and, other anti-patterns identified as non-faulty and participating in Asynchrony change patterns with other files.

The results of the Fisher's exact tests and odds ratios when testing H_{RQ1_0} are significant. For all the analyzed systems, the p-value is less than 0.05 and the likelihood that an anti-pattern that shares an Asynchrony change pattern with other anti-patterns experiences a fault is about two to four times higher than the likelihood that other anti-patterns experience faults if they share an Asynchrony change pattern with other files.

We can also answer positively **RQ1** as follows: anti-patterns that share Asynchrony change patterns with other anti-patterns are significantly more fault-prone than anti-patterns that share Asynchrony change patterns with other files.

5.2. What is the impact on fault-proneness when a file participates both in clones and Asynchrony change pattern occurrences?

5.2.1. Motivation

Co-change between code clones is considered in some previous work as a threat to software quality and maintenance, because, on the one hand, faults may have been cloned in parallel with the cloned co-changed code. On the other hand, it is expected that changes done to the original code must often be applied to the cloned code as well.

However, there exist only few empirical studies that analyze clones that follow similar change patterns. Knowing the relations among clones and change patterns and their impact on fault-proneness assists programmers to discover files that must be checked to guarantee the maintainability of software systems and decrease the menace of faults.

Table 3: Fault proneness and Fisher's test results on ArgoUML, JFreeChart, Lucene, and XalanJ for clones participating in Asynchrony change patterns (CACPC: Clones participating in Asynchrony change patterns with clones, CACPNC: Clones that follow Asynchrony change patterns with other files but not clones)

	Faulty	Non-faulty
Number of CACPC in ArgoUML	130	6
Number of CACPNC in ArgoUML	37	7
Fisher's test for ArgoUML	0.01751	
Odds-ratio for ArgoUML	4.058749	
Number of CACPC in JFreeChart	30	59
Number of CACPNC in JFreeChart	12	52
Fisher's test for JFreeChart	0.03014	
Odds-ratio for JFreeChart	2.192315	
Number of CACPC in Lucene	17	25
Number of CACPNC in Lucene	6	26
Fisher's test for Lucene	0.0389	
Odds-ratio for Lucene	2.904332	
Number of CACPC in XalanJ	43	1
Number of CACPNC in XalanJ	12	3
Fisher's test for XalanJ	0.04699	
Odds-ratio for XalanJ	10.20738	

5.2.2. Method

For each software system, we identify files that contain clones using CCFinder. Second, we identify clones involved in an Asynchrony change pattern using Macocha. Then, we identify if a file undergoes faults during the same period as described in Section 3. Finally, we verify the null hypothesis which we state as follows:

- H_{RQ2_0}: There is no statistically significant difference between proportion of faults carried by clones participating in Asynchrony change pattern with clones and clones that share Asynchrony change patterns with other files in ArgoUML and JFreeChart.

5.2.3. Results

Table 3 presents the fault proneness analysis for ArgoUML, JFreeChart, Lucene, and XalanJ that reports on the results of Fisher's exact test and odds ratios when testing H_{RQ2_0} are significants. The p-value is less than 0.05

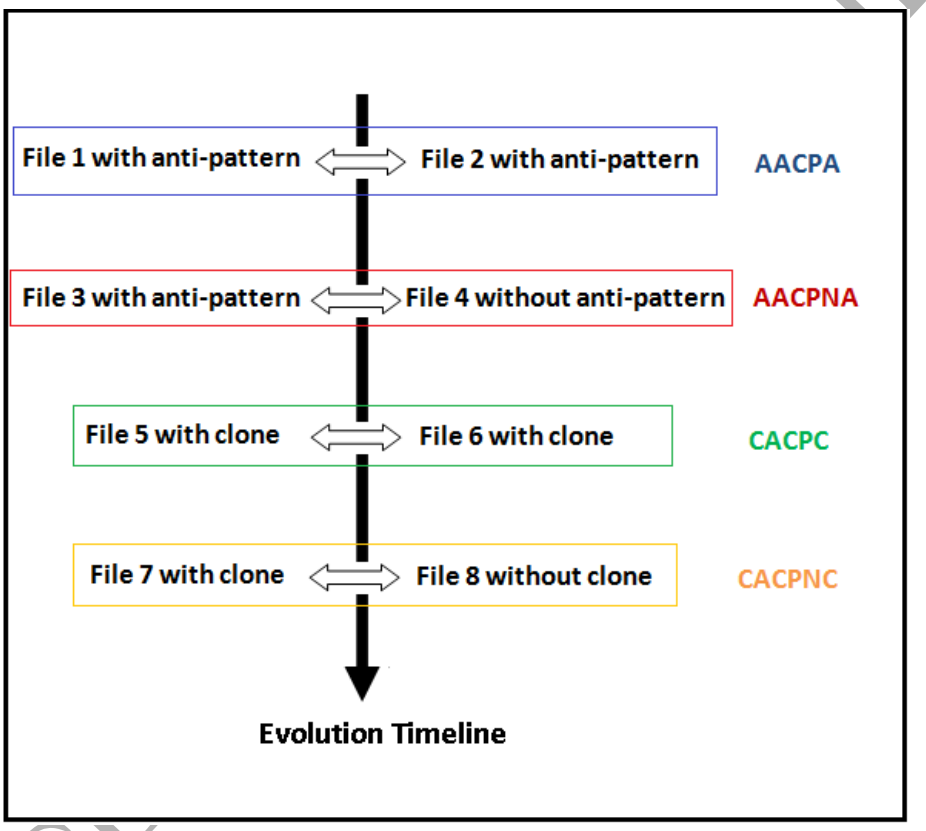

Figure 4: The analysed sets in this study

and. For example, in ArgoUML, the odds ratio for fault-prone clones that adhere to Asynchrony change patterns with clones is four times higher than clones that adhere to Asynchrony change patterns with other files. We can thus answer to **RQ2** as follows: we found that clones sharing Asynchrony change pattern are significantly more fault-prone than other clones. Sharing Asynchrony change patterns among clones thus indeed correlates with the fault proneness of files.

5.3. What is the fault-proneness of files that participate in anti-patterns and clones?

5.3.1. Motivation

Previous work showed that some smells are more fault prone than others [1] [51] [2]. Entities affected by anti-patterns and clones co-occurrence could have a high risk of fault-proneness. Thus, they must be checked in depth to guarantee a fine maintainability of software systems and decrease the risk of faults in the future. We are attempting to compare the fault proneness across the following groups of files: files belonging to an anti-pattern, files having clones, files belonging to an anti-pattern and having clones, and files not belonging to an anti-pattern and not having clones.

5.3.2. Approach

For each system, we identify whether a file has faults during the analysed period of the studied systems as described in Section 3.

Then, we use Chi-squared's test [61] to check whether the difference in fault-proneness between files sharing the considered anti-patterns and clones and other files is significant. We test the following null hypothesis:

- H_{RQ3_0}: The proportions of faults involving files having a co-occurrence of anti-patterns and code clones and other files are the same. If we reject the null hypothesis H_{RQ3_0}, the proportion of faults carried by files having a co-occurrence of anti-patterns and code clones is not the same as that of other files.

5.3.3. Results

Table 4 presents fault proneness analysis for ArgoUML, JFreeChart, Lucene, and XalanJ. Indeed, the result of Chi-squared's test and odds ratios when testing H_{RQ3_0} are significant, the p-value is less than 0.05 and the odds ratio for fault-prone CAC files (files presenting co-occurrence of anti-pattern and clone) is higher than for fault-prone other files.

Table 4: Fault proneness and Chi-squared's test results in ArgoUML, JFreeChart, Lucene, and XalanJ (CAC: presenting co-occurrence of anti-pattern and clone)

	Faulty	Non-faulty
Number of files CAC in ArgoUML	126	41
Number of the rest of files in ArgoUML	992	2166
Chi-squared's test for ArgoUML	0.0001	
Odds-ratio for ArgoUML	6.7102	
Effect size for ArgoUML	0.204	
Number of files CAC in JFreeChart	31	8
Number of the rest of files in JFreeChart	578	998
Chi-squared's test for JFreeChart	0.0001	
Odds-ratio for JFreeChart	6.6907	
Effect size for JFreeChart	0.136	
Number of files CAC in Lucene	15	4
Number of the rest of files in Lucene	289	442
Chi-squared's test for Lucene	0.0013	
Odds-ratio for Lucene	6.022	
Effect size for Lucene	0.126	
Number of files CAC in XalanJ	42	13
Number of the rest of files in XalanJ	331	681
Chi-squared's test for XalanJ	9.977e-11	
Odds-ratio for XalanJ	5.8613	
Effect size for XalanJ	0.202	

Indeed, we can answer **RQ3** as follows: we showed that classes having clones and anti-patterns are significantly more fault-prone than other classes.

6. Discussion and Threats to Validity

We now discuss several observations from our results. Then, we discuss the threats to validity of our study.

6.1. Impact of Smell Co-occurrence on Software Quality

The fault-proneness is one of the metrics that are used to measure the quality of software systems. A fault is a manifestation of an error in software. In accordance with the IEEE Standard Classification for Software Anomalies, a defect is the algorithmic error in a program that may lead to a fault. Indeed,

a defect is a fault if it is encountered during software execution. Thus, in order to evaluate the quality of systems, many previous works used the number of faults per line as a quality measure [65, 66]. In the same context, we notice that in the two systems considered in this study, files having co-occurrences of clone and anti-pattern are more fault-prone than other files (see Table 4). For example, in ArgoUML, files with co-occurrences of clones and anti-patterns are 6.7 times more fault-prone than files that do not have clone and anti-pattern and still higher even if a class had an occurrence of an anti-pattern or a clone. One possible interpretation for this result is that copying code smells can increase the fault injection and the risk of introducing unknown types of faults in the classes. Many previous studies have observed a relation between the number of lines of code and the fault-proneness. Indeed, when a system had a high number of lines of code, it showed a larger number of smell co-occurrence and thus the worst fault-proneness degree. This observation suggested that producing too large files increases the co-occurrence probability of clones and anti-patterns and thus degrades software quality. However, some other studies showed that smaller files had disproportionally many fault, as noticed for example by Koru *et al.* [67]. Finally, we observed that anti-patterns resulting in large classes (classes that are involved in anti-patterns like LargeClass or Blob), are not the most affected by clones in the considered systems. This observation rejects the possible impact of the size of files measured by the LOC metric on the fault-proneness of classes with co-occurrence of clones and anti-patterns.

6.2. Smell Co-occurrence, Change and Fault Propagation

Our previous studies analyzed independently the impact of anti-patterns [5] and clones [6] on change propagation on fault-proneness. We thus reported that the detection and the analysis of anti-patterns and clones allow us to detect the critical elements of software systems since they may represent the set of more risky files in terms of fault-proneness. Other previous work showed that clones require consistent changes [68] [59]. A consistent change is a change that alters in the same way and at the same time, all instances of a clone group. We also observed in a previous paper [5] the existence of change dependencies between anti-patterns and other artefacts in software systems. If these dependencies are not properly maintained, they can increase the risk of faults. Such risks may increase in the case of co-occurrence of anti-patterns and clones as several previous works have shown the negative impact of anti-patterns on fault-proneness. Thus, by spotting the sets of co-occurrences of

anti-patterns and clones, we identify the more risky part of code in term of consistent change.

6.3. Threats to Validity

We now discuss in detail the threat to validity of our results, following the guidelines provided in [69].

Construct validity threats concern the relation between theory and observation. In our context, they are mainly due to the definition and the measure of anti-patterns, clones and change patterns. We are aware that the detection technique used includes some subjective understanding of the definition of smells. For this reason, the precision values of the change pattern, anti-pattern and clone detection tools is a concern that we agree to accept. In case that pattern specifications would be slight variants of the specifications used by Macocha, CCFinder and DECOR, some change pattern, clone or anti-pattern occurrences may be missed during the detection phase. Although the sample of detected patterns used in our study can be considered large enough to claim our conclusions valid, additional investigations are desirable to further verify our findings with other tools or other definitions of patterns.

Reliability validity threats concern the possibility of replicating this study. First, the source code repositories of the analysed systems are publicly available. Then, the anti-pattern, change pattern and clone detection tools used in this study could be used freely. Finally, the change logs, the list of bugs and the changed files of the analysed programs with the list of clones and anti-pattern occurrences to obtain our observations are available on-line at `http://www.ptidej.net/downloads/experiments/JSS2016/`.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate the assumptions of the statistical test that we used, *i.e.*, the Fisher's exact test, which is a non-parametric test. A possible threat to the conclusion validity is our particular choice of anti-patterns. Although these anti-patterns are and have been widely analyzed by other researchers, there is no consensus on their universality. Another possible threat to the conclusion validity is the possible interference between the clone dataset and the anti-pattern dataset. We can find a file that has at the same time an anti-pattern and clone. In our study, we manually validates such cases and we disagreed them from our analysis to be sure that such interference will not have an impact on the interpretation of the results. However, we analyze the resulting impact of the interference of clones and anti-patterns in section 5.

Threats to *external validity* concern the possibility to generalize our observations. Different systems with different CVS/SVN change logs, requirements, and contexts could lead to different results. However, the two selected datasets have different SVN change logs, requirements, and source code quality. Our dataset selection thus reduces this threat to external validity. However, more studies, for example on industrial datasets, are required to generalize the results of our empirical study.

7. Conclusion

Several previous works analysed the impact of anti-patterns or cloned code on fault proneness. Assuming that change propagation (as described by Asynchrony change pattern) can increase the fault proneness of risky part of code (such as clones and anti-patterns), we reported in this paper an empirical study investigating the fault-proneness in ArgoUML and JFreeChart of anti-patterns and clones involved in Asynchrony change patterns.

Our study's results show that, in all systems, the fault proneness odds ratios significantly increase for anti-patterns and clones participating in the Asynchrony change pattern occurrences. We found also that file fault proneness odds ratios significantly increase for files that had co-occurrence of anti-patterns and clones in comparison with the rest of files in the studied systems. Thus, we empirically found a specific and tangible correlation of clones and anti-patterns on software quality when they are not properly maintained.

To confirm the conjectures established with this study, we are working in several extensions of this work. Among others: the replication of the study at class level instead of file level (by adding the analysis of C++ software systems for example) and the prediction of the co-occurrence of clones and anti-patterns and its impact on software quality.

Acknowledgment

This work has been partly funded by NSERC and the Research and Development Grants Quebec-Wallonie.

References

[1] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Software Engineering (2012) 243–275.

[2] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter?, in: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2009, pp. 485–495.

[3] S. Bouktif, G. Antoniol, E. Merlo, M. Neteler, A plugin based architecture for software maintenance, Tech. Rep. EPM-RT-2006-03, Department of Computer Science École Polytechnique de Montréal (2006).

[4] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, Analysing anti-patterns static relationships with design patterns, Journal of Electronic Communications of the European Association of Software Science and Technology (accepted).

[5] F. Jaafar, Y.-G. Gueheneuc, S. Hamel, F. Khomh, Mining the relationship between anti-patterns dependencies and fault-proneness, in: Reverse Engineering (WCRE), 2013 20th Working Conference on, IEEE, 2013, pp. 351–360.

[6] A. Lozano, F. Jaafar, K. Mens, Y. Guéhéneuc, Clones and macro co-changes, Electronic Communications of the European Association of Software Science and Technology 63 (2014) 1–14.

[7] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, G. Antoniol, Detecting asynchrony and dephase change patterns by mining software repositories, Journal of Software: Evolution and Process 26 (1) (2014) 77–106.

[8] R. Geiger, B. Fluri, H. C. Gall, M. Pinzger, Relation of code clones and change couplings, in: Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 411–425.

[9] A. Yamashita, L. Moonen, To what extent can maintenance problems be predicted by code smell detection? - an empirical study, Information Software Technology 55 (12) (2013) 2223–2242.

[10] A. T. T. Ying, G. C. Murphy, R. Ng, M. C. Chu-Carroll, Predicting source code changes by mining change history, Transactions on Software Engineering 30 (9) (2004) 574–586.

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

[12] M. Ceccarelli, L. Cerulo, G. Canfora, M. Di Penta, An eclectic approach for change impact analysis, in: Proceedings of the 32nd International Conference on Software Engineering, ACM Press, New York, NY, USA, 2010, pp. 163–166.

[13] G. Canfora, M. Ceccarelli, L. Cerulo, M. Di Penta, Using multivariate time series and association rules to detect logical change coupling: An empirical study, in: Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Washington, DC, USA, 2010, pp. 1–10.

[14] B. F. Webster, Pitfalls of Object Oriented Development, 1^{st} Edition, M & T Books, 1995.

[15] A. J. Riel, Object-Oriented Design Heuristics, 1st Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[16] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, T. J. Mowbray, Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1^{st} Edition, John Wiley and Sons, 1998.

[17] E. V. Emden, L. Moonen, Java quality assurance by detecting code smells, in: The 9th Working Conference on Reverse Engineering. IEEE Computer, Society Press, 2002, pp. 97–107.

[18] D. Ratiu, S. Ducasse, T. Gîrba, R. Marinescu, Using history information to improve design flaws detection, in: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2004, pp. 223–233.

[19] B. S. Baker, On finding duplication and near-duplication in large software systems, in: Reverse Engineering, 1995., Proceedings of 2nd Working Conference on, IEEE, 1995, pp. 86–95.

[20] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Proceedings of the IEEE International

Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 1999, pp. 109–118.

[21] R. Koschke, S. Bazrafshan, Software-clone rates in open-source programs written in c or c++, in: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 3, 2016, pp. 1–7.

[22] W. T. Cheung, S. Ryu, S. Kim, Development nature matters: An empirical study of code clones in javascript applications, Empirical Software Engineering 21 (2) (2016) 517–564.

[23] C. Li, J. Sun, H. Chen, An improved method for tree-based clone detection in web applications, in: Digital Information and Communication Technology and it's Applications (DICTAP), Fourth International Conference on, IEEE, 2014, pp. 363–367.

[24] M. Fowler, Refactoring: Improving the design of existing code, International Thomson Computer Press, 1999.

[25] C. Kapser, M. W. Godfrey, " cloning considered harmful" considered harmful, in: Reverse Engineering, 2006. WCRE'06. 13th Working Conference on, IEEE, 2006, pp. 19–28.

[26] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on, IEEE, 1999, pp. 109–118.

[27] S. Lee, I. Jeong, Sdd: high performance code clone detection system for large scale source code, in: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM, 2005, pp. 140–141.

[28] J. R. Cordy, T. R. Dean, N. Synytskyy, Practical language-independent detection of near-miss clones, in: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, 2004, pp. 1–12.

[29] R. Koschke, R. Falke, P. Frenzel, Clone detection using abstract syntax suffix trees, in: Reverse Engineering, 2006. WCRE'06. 13th Working Conference on, IEEE, 2006, pp. 253–262.

[30] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, M. Bernstein, Pattern matching for clone and concept detection, in: Reverse engineering, Springer, 1996, pp. 77–108.

[31] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, K. Kontogiannis, Measuring clone based reengineering opportunities, in: Software Metrics Symposium, 1999. Proceedings. Sixth International, IEEE, 1999, pp. 292–303.

[32] W. Yang, Identifying syntactic differences between two programs, Software: Practice and Experience 21 (7) (1991) 739–755.

[33] R. Falke, P. Frenzel, R. Koschke, Empirical evaluation of clone detection using syntax suffix trees, Empirical Software Engineering 13 (6) (2008) 601–643.

[34] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Software Maintenance, 1998. Proceedings., International Conference on, IEEE, 1998, pp. 368–377.

[35] R. Komondoor, S. Horwitz, Effective, automatic procedure extraction, in: Program Comprehension, 2003. 11th IEEE International Workshop on, IEEE, 2003, pp. 33–42.

[36] J. Krinke, Identifying similar code with program dependence graphs, in: Reverse Engineering, 2001. Proceedings. Eighth Working Conference on, IEEE, 2001, pp. 301–309.

[37] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Transactions on software engineering 33 (9).

[38] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: a multilinguistic token-based code clone detection system for large scale source code, Vol. 28, IEEE, 2002.

[39] T. J. McCabe, A complexity measure, in: Proceedings of the 2Nd International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA, 1976, pp. 407–417.

[40] M. H. Halstead, Elements of Software Science (Operating and Programming Systems Series), Elsevier Science Inc., New York, NY, USA, 1977.

[41] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transaction On Software Engineering 20 (6) (1994) 476–493.

[42] V. R. Basili, L. C. Briand, W. L. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Transaction On Software Engineering 22 (10) (1996) 751–761.

[43] R. Subramanyam, M. S. Krishnan, Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects, IEEE Transaction On Software Engineering 29 (4) (2003) 297–310.

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

[45] M. D'Ambros, M. Lanza, R. Robbes, On the relationship between change coupling and software defects, in: Proceedings of the 16th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, 2009, pp. 135–144.

[46] A. Marcus, D. Poshyvanyk, R. Ferenc, Using the conceptual cohesion of classes for fault prediction in object-oriented systems, IEEE Transactions On Software Engineering (2008) 287–300.

[47] T. Ostrand, E. Weyuker, R. Bell, Predicting the location and number of faults in large software systems, IEEE Transactions on Software Engineering (2005) 340–355.

[48] A. Bernstein, J. Ekanayake, M. Pinzger, Improving defect prediction using temporal features and non linear models, in: Ninth International Workshop on Principles of Software Evolution, ACM, 2007, pp. 11–18.

[49] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: The 14th Conference on Computer and Communications Security, ACM, 2007, pp. 529–540.

[50] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: The 30th International Conference on Software Engineering, ACM, New York, NY, USA, 2008, pp. 181–190.

[51] L. Aversano, L. Cerulo, M. Di Penta, How clones are maintained: An empirical study, in: Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on, IEEE, 2007, pp. 81–90.

[52] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur, DECOR: A method for the specification and detection of code and design smells, Transactions on Software Engineering (2010) 20–36.

[53] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes?, SIGSOFT Software Engineering Notes 30 (4) (2005) 1–5.

[54] Y.-G. Guéhéneuc, G. Antoniol, Demima: A multilayered approach for design pattern identification, IEEE Transactions on Software Engineering 34 (5) (2008) 667–684.

[55] Y.-G. Guéhéneuc, H. Sahraoui, F. Zaidi, Fingerprinting design patterns, in: Reverse Engineering, 2004. Proceedings. 11th Working Conference on, IEEE, 2004, pp. 172–181.

[56] D. Romano, P. Raila, M. Pinzger, F. Khomh, Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes, Working Conference on Reverse Engineering (2012) 437–446.

[57] E. Burd, J. Bailey, Evaluating clone detection tools for use during preventative maintenance, in: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society, Washington, DC, USA, 2002, pp. 36–46.

[58] D. Rattan, R. Bhatia, M. Singh, Software clone detection: A systematic review, Information and Software Technology 55 (7) (2013) 1165–1199.

[59] A. Lozano, M. Wermelinger, Assessing the effect of clones on changeability, in: Proceeding of 24th International Conference on Software Maintenance, IEEE, 2008, pp. 227–236.

[60] BasiliV., PerriconeB.T., Software errors and complexity: An empirical investigation, Communications of the ACM 27 (1) (1984) 42–52.

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

[62] T. Pedersen, Fishing for exactness, In Proceedings of the South-Central SAS Users Group Conference cmp-lg/9608010 (1996) 188–200.

[63] D. Settas, A. Cerone, S. Fenz, Enhancing ontology-based antipattern detection using bayesian networks, Expert Systems with Applications 39 (10) (2012) 9041–9053.

[64] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2011, pp. 181–190.

[65] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, K.-i. Matsumoto, Software quality analysis by code clones in industrial legacy software, in: Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on, IEEE, 2002, pp. 87–94.

[66] M. Ohba, Software reliability analysis models, IBM Journal of research and Development 28 (4) (1984) 428–443.

[67] A. G. Koru, D. Zhang, K. El Emam, H. Liu, An investigation into the functional form of the size-defect relationship for software modules, IEEE Transactions on Software Engineering 35 (2) (2009) 293–304.

[68] J. Krinke, A study of consistent and inconsistent changes to code clones, in: Proceedings of the 14th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, 2007, pp. 170–178.

[69] R. K. Yin, Case Study Research: Design and Methods - Third Edition, SAGE Publications, London, 2002.

Fehmi Jaafar is an Adjunct Professor at the Faculty of Management at Concordia University of Edmonton and a Research Scientist in Computer Security and Software Engineering at the Research And Development Team of Ubitrak Inc. He received his Ph.D. from the Department of Computers Sciences and Operations Research of the Université de Montréal in Quebec, Canada, 2013. His research interests include: Software Quality and Evolution, Mining Software Repositories, and Fault Proneness.

Angela Lozano is a researcher interested on supporting software engineers to implement changes in evolving applications. Currently, she is working on a description of source code entities at a very fine grain that is resilient to changes. In particular, her PhD project is a method to assess the effect of source code characteristics on the evolvability of an implementation. The method proposed was evaluated by analyzing the effect of clones on the changeability of methods.

Yann-Gael Gueheneuc is a full professor at the Department of computer and software engineering of Polytechnique de Montreal where he leads the Ptidej team on evaluating and enhancing the quality of object- oriented programs by promoting the use of patterns, at the language-, design-, or architectural-levels. He is IEEE Senior Member since 2010. In 2009, he was awarded the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointes supervision) since 2003 and an Engineering Diploma from Ecole des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns.

Kim Mens is full Professor in Computer Science at the Universit catholique de Louvain (UCL), Belgium where he leads the REsearch Laboratory on software Evolution And Software Development technology (RELEASeD). He holds the degrees of Licentiate in Mathematics, Licentiate in Computer Science, and PhD in Computer Science from the Vrije Universiteit Brussel (VUB), Belgium. His main research interests include software development, software maintenance and software evolution with a particular emphasis on the programming aspect and tool support. Other research topics that fit this common theme and in which he is interested are software architecture, software variability, reverse engineering, software transformation, software

restructuring and renovation, and co-evolution between source code and earlier lifecycle software artifacts. In addition to that he is interested in language engineering and a variety of programming language paradigms such as object-oriented and aspect-oriented programming, reflection and metaprogramming, logic metaprogramming and, more recently, context-oriented programming.