Empir Software Eng (2014) 19:1–38
DOI 10.1007/s10664-012-9214-z

How changes affect software entropy: an empirical study

Gerardo Canfora · Luigi Cerulo · Marta Cimitile · Massimiliano Di Penta

Published online: 14 July 2012

Editor: Sandro Morasca

Abstract Software systems continuously change for various reasons, such as adding new features, fixing bugs, or refactoring. Changes may either increase the source code complexity and disorganization, or help to reducing it. This paper empirically investigates the relationship of source code complexity and disorganization—measured using source code change entropy—with four factors, namely the presence of refactoring activities, the number of developers working on a source code file, the participation of classes in design patterns, and the different kinds of changes occurring on the system, classified in terms of their topics extracted from commit notes. We carried out an exploratory study on an interval of the life-time span of four open source systems, namely ArgoUML, Eclipse-JDT, Mozilla, and Samba, with the aim of analyzing the relationship between the source code change entropy and four factors: refactoring activities, number of contributors for a file, participation of classes in design patterns, and change topics. The study shows that (i) the change entropy decreases after refactoring, (ii) files changed by a higher number of

This paper is an extension of the paper "An Exploratory Study of Factors Influencing Change Entropy" (Canfora et al. 2010).

G. Canfora · M. Di Penta (✉)
Department of Engineering-RCOST, University of Sannio, Benevento, Italy
e-mail: dipenta@unisannio.it
URL: www.rcost.unisannio.it/mdipenta

G. Canfora
e-mail: canfora@unisannio.it
URL: www.gerardocanfora.net/

L. Cerulo
Department of Biological and Environmental Studies, University of Sannio, Benevento, Italy
e-mail: lcerulo@unisannio.it
URL: http://rcost.unisannio.it/cerulo

M. Cimitile
Department of Jurisprudence, Unitelma Sapienza, Napoli, Italy
e-mail: marta.cimitile@unitelma.it

developers tend to exhibit a higher change entropy than others, (iii) classes participating in certain design patterns exhibit a higher change entropy than others, and (iv) changes related to different topics exhibit different change entropy, for example bug fixings exhibit a limited change entropy while changes introducing new features exhibit a high change entropy. Results provided in this paper indicate that the nature of changes (in particular changes related to refactorings), the software design, and the number of active developers are factors related to change entropy. Our findings contribute to understand the software aging phenomenon and are preliminary to identifying better ways to contrast it.

Keywords Software entropy · Software complexity · Mining software repositories

1 Introduction

During their lifetime, software systems undergo to maintenance activities, having different purposes, such as introducing new features to comply with user needs or make the software competitive with respect to an alternative product, fixing faults, or adapting the software to new environments and architectures. Due to time pressure, limited resources, and the lack of a disciplined maintenance process, these activities tend to deteriorate the software system structure, thus increasing source code complexity and, in general, making the system more difficult to be understood and maintained in the future. Parnas (1994) calls this phenomenon "software aging":

> Like human aging, software aging is inevitable, but like human aging, there are things that we can do to slow down the process and, sometimes, even reverse its effects.

Lehman and Belady, in their second law of software evolution (Lehman 1980; Lehman and Belady 1985), state that:

> As an E-type system evolves its complexity increases, unless work is done to maintain or reduce it.

The Lehman and Belady law, in addition to making a statement on the increasing complexity of a software system, points out that there can be countermeasures to mitigate such a deterioration. Reengineering, redocumentation (Chikofsky and Cross 1990) and refactoring (Fowler et al. 1999) are few examples of activities aimed at mitigating, or even reversing, the effects of aging. In summary, the evolution of a software system can be characterized by different kinds of changes, some of them increasing the software complexity, some decreasing it, and some having a limited impact.

This paper investigates the relationships between the complexity of a software system and factors that could influence it. To quantify the complexity and disorganization of the source code, this paper uses previous work (Bianchi et al. 2001; Chapin 1995; Harrison 1992; Hassan 2009) that adapted Shannon's entropy (Shannon 1948) to measure the complexity of an evolving Software system. Specifically, we use the Hassan (2009) definition of change entropy, which considers a change to introduce

more entropy when it affects many files: the rationale is that developers find it more difficult to keep track of changes scattered across many files, and, as Hassan empirically showed, files with a higher change entropy tend to exhibit a higher fault-proneness. We investigate four different factors that can be related to source code change entropy, specifically:

- to what extent would a refactoring activity reduce the entropy of future changes occurring on files on which the refactoring was performed;
- whether the change entropy of a file is related to the number of developers that changed the file;
- whether the participation of classes in specific kinds of design patterns correlates with a higher (or lower) change entropy than in other classes; and
- whether different kinds of changes entail different change entropy. Specifically, we use an information retrieval technique, the Latent Dirichlet Allocation (LDA) (Blei et al. 2003) to identify topics in versioning system commit notes describing the changes, and classify changes according to such topics.

We considered these factors as they account for kinds of changes that can increase or decrease the change entropy—e.g., changes related to refactorings or to specific topics—to the relation between entropy and the number of developers modifying a file (also considered as a software complexity factor by Eick et al. 2001), and to design structures—design patterns in particular—that can influence the change entropy positively or negatively.

The study analyzed the history of four open source systems, ArgoUML (Java), Eclipse-JDT (Java), Mozilla (C/C++), and Samba (C). Results show that (i) as expected, refactorings mitigate the change entropy, i.e., changes occurring on a file after it underwent a refactoring exhibit a lower change entropy than changes occurring before the refactoring; (ii) files modified by a higher number of developers exhibit a higher change entropy; (iii) classes participating in certain design patterns, exhibit a lower change entropy than others, though this depends on the context in which the design pattern is used in a given system; and (iv) different kinds of changes exhibit different change entropy. For example, focused changes, such as bug fixings of copyright changes exhibit a low change entropy. Instead refactorings, by themselves, exhibit a high change entropy, while changes following such refactorings exhibit a decreased change entropy.

The paper is organized as follows. Section 2 reviews the definition of software entropy provided in literature and introduces the definition of change entropy used in this paper. Section 3 defines the empirical study providing details about its research questions, variables, data extraction, and analysis method. Results are reported and discussed in Section 4. Section 5 discusses the threats to validity. After a discussion of the related literature (Section 6), Section 7 concludes the paper.

2 Software Entropy

This section introduces the concept of software entropy, and then specifically recalls the definition of software change entropy as introduced by Hassan (2009).

Change entropy derives from Shannon entropy, a measure of the uncertainty associated with a random variable which quantifies the information contained in a

message produced by a *data emitter* (Shannon 1948). For a discrete random variable X with n possible outcomes, $x_1, x_2, \ldots, x_n$, the entropy of X is defined as: $H(X) = -\sum_{i=1}^{n} p(x_i)\, log_2\, p(x_i)$, where $p(x_i)$ is the probability mass function of the outcome x_i. If the outcomes have the same probability of occurrence, i.e., $p(x_i) = 1/n$, the entropy is maximal; instead, for a distribution X where the outcome x_k has a probability of occurrence $p(x_k) = 1$, and all other $p(x_i) = 0$, then the entropy is minimal ($H(X) = 0$). The Shannon's definition of entropy can be interpreted as the number of bits needed to uniquely identify a data distribution: the higher the entropy the higher the *uncertainty* in identifying the distribution.

The basic code change model proposed by Hassan (2009) considers a software system as a *data emitter*, and source file modifications as the data generated by the emitter. In particular, given a software system $S \equiv \{f_1, f_s, .., f_n\}$ composed of a set of files f_i, the random variable is the software system S, and the outcome is the modification performed to a source file $f_i \in S$.

The entropy of a change activity—e.g., of a cohesive set of changes aimed at implementing a feature or at fixing a bug—in which each file f_i underwent a certain number of changes $chg(f_i)$, is defined as:

$$H(S) = -\sum_{i=1}^{n} \frac{chg(f_i)}{chg(S)} \log_2 \left(\frac{chg(f_i)}{chg(S)} \right) \tag{1}$$

The probability mass function of f_i, i.e., the probability that a source file changes in such an activity, is estimated as the ratio between $chg(f_i)$ and the total number of changes in the activity.

A relevant question is what has to be considered a change activity, or, equivalently, the period over which the change entropy should be computed. There are different possibilities:

- *a time interval*, e.g., a week;
- *a given number of commits*, which was the approach we followed in our previous paper (Canfora et al. 2010), where we considered a change as a sequence of 500 commits;
- *a change set*, defined as a sequence of file revisions that share the same author, branch, and commit notes, and such that the difference between the timestamps of two subsequent commits is less or equal than 200 s (Zimmermann et al. 2004).

When computing the entropy, other units of code, such as classes or functions, can be used instead of source files. The choice of source file is based on the belief that it is a conceptual unit of development where developers tend to group strongly coupled entities. For example, a Java source code file often contains a class, while a C file often contains functions belonging to a module. Such a choice makes our analysis independent of the programming language—e.g., procedural vs. object-oriented.

Figure 1 shows an example of how the change entropy of a file is computed. The system comprises four source code files, and 10 changes were performed for all those files in the gray-shaded period. f_A and f_B were modified once, so we have a $p(f_A) = p(f_B) = 1/10$, f_C was modified three times, then we get $p(f_C) = 3/10$, and f_D was modified five times, then we get $p(f_D) = 5/10$. On the right side of the figure, a graph shows the file change probability distribution for the gray-shaded period.

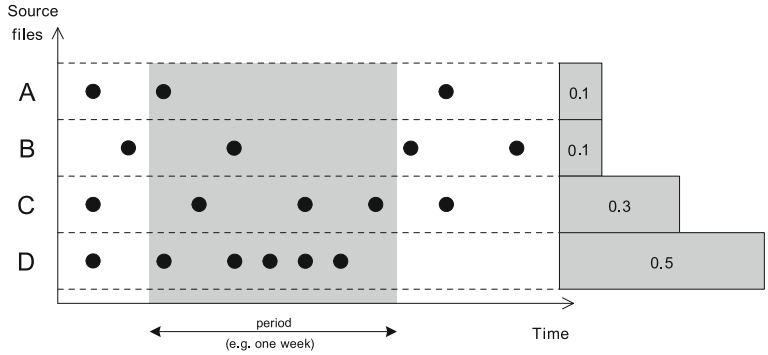


Fig. 1 Complexity of a change period (Hassan 2009). It is computed (*right-side bars*) as the number of changes to a file in a period divided by the total number of changes in the period

Change entropy within a period of time increases when a change is scattered across many source files, instead it reaches a minimum when a change is performed to a single source file. As stated by Hassan (2009), the definition of change entropy is directly related with the intuition that developers will have a harder work keeping track of changes that are performed across many source files.

The entropy defined in (1) does not account for the varying number of files in a software system. Thus, this definition is not suitable to analyze the timeframe during which new files are added and/or existing files are removed. Hassan (2009) proposed a way to normalize the entropy as follows:

$$H(S) = -\sum_{i=1}^{1} \frac{chg(f_i)}{chg(S)} \log_n \left(\frac{chg(f_i)}{chg(S)} \right) \tag{2}$$

where n is the number of files in the system when snapshot S has been extracted, which can vary as the system evolves.

To keep into account to what extent a file was affected by a change, and thus avoiding to weight in the same way changes affecting one line and changes affecting most of the file, the probability mass function of f_i can be computed as the ratio, $chglines(f_i)/chglines(S)$, i.e. the number of lines (added/removed/changed) affected by the change in file f_i, and the total number of lines affected by the change. Thus, the *line-based entropy* becomes:

$$H(S) = -\sum_{i=1}^{n} \frac{chg(f_i)}{chg(S)} \log_{nlines} \left(\frac{chglines(f_i)}{chglines(S)} \right) \tag{3}$$

where *nlines* is the number of lines in the system at the time of snapshot S. The use of $\log_{nlines}$ instead of $\log_2$ is, again, used to normalize the change entropy.

To analyze the change entropy of a specific file, Hassan proposed a measure that weights the entropy of a change activity $H'(S)$ with the change probability of f_i during such activity, i.e.,

$$H'(f_i) \equiv H'(S) \cdot \frac{chg(f_i)}{chg(S)} \tag{4}$$

which is zero when a file does not change within the change activity. In terms of changed lines, it is computed as:

$$H'(f_i) \equiv H'(S) \cdot \frac{chglines(f_i)}{chglines(S)} \tag{5}$$

where $chglines(f_i)$ is the number of lines (added/removed/changed) affected by the change in file f_i, and $chglines(S)$ is the total number of lines affected by the change. Hassan showed that the file change entropy is correlated to fault-proneness (Hassan 2009).

3 Empirical Study

The *goal* of this study is to analyze how factors related to software characteristics and development activities—specifically the occurrence of refactoring changes, the number of committers that changed a file, the participation of classes into design patterns, and the change topics—relate with change entropy. The *quality focus* is the source code maintainability and comprehensibility, that could be affected by the increasing change entropy. The *perspective* is of researchers interested to investigate to what extent the change entropy can be controlled by means of refactoring activities, or by limiting the number of people modifying a file. The *context* consists of change data from four open source systems, having a different size and developed with different languages: ArgoUML (medium, Java), Eclipse-JDT (large, Java), Mozilla (large, C/C++), and Samba (medium, C).

ArgoUML[1] is an open source UML modeling tool with advanced software design features, such as reverse engineering and code generation. Eclipse-JDT is a set of plug-ins that adds the capabilities of a full-featured Java IDE to the Eclipse[2] platform. *Mozilla*[3] is a suite comprising a Web browser, an email client, and other Internet utilities in C/C++. Samba[4] is a software suite, written in ANSI C, that provides file and print services and allows for interoperability between Linux/Unix servers and Windows-based clients.

Table 1 shows the main characteristics of the four systems, including (i) the time interval we analyzed, (ii) the minimum and maximum number of files and KNLOC (non commented KLOC), (iii) the total number of file changes (commits) considered, (iv) the number of change sets extracted from the Concurrent Versions Systems (CVS)/SubVersioN (SVN) repositories, (v) the number of change sets in which a refactoring activity was performed (and documented), and (vi) the total number of project committers.

[1] http://argouml.tigris.org

[2] http://www.eclipse.org

[3] http://www.mozilla.org

[4] http://www.samba.org

Table 1 Characteristics of the analyzed projects

System	Analyzed period	Files (range)	KNLOC (range)	Commits	Change sets	Refact. change sets	Committers
ArgoUML	01/1998–03/2009	863–1,982	66–210	41,229	4,040	530	37
Eclipse-JDT	05/2001–05/2011	2,089–4,321	205–733	39,497	6,687	338	52
Mozilla	03/1998–02/2011	4,845–12,358	1,827–3,169	171,625	31,891	435	628
Samba	05/1996–04/2004	56–889	36–345	8,151	2,326	438	25

3.1 Research Questions

The research questions this study aims at addressing are the following:

- **RQ1:** *How do refactorings affect the change entropy?* This research question aims at investigating whether a refactoring-related change contributes to decreasing (or increasing) the entropy of future changes. The conjecture we would like to empirically investigate is that, after a refactoring, the number of files a developer has to change, to perform a maintenance activity, would, on average, decrease. The null hypothesis being tested is H_{01}: *there is no significant difference between the average entropy of changes occurring before and after a refactoring activity*.
- **RQ2:** *How does the change entropy relate to the number of contributors to a file?* This research question aims at investigating whether the change entropy of a file is correlated to the number of developers modifying it. The conjecture is that, on the one hand, files modified by more persons would exhibit a higher change entropy because, for example, a developer could wrongly modify the source code written by someone else, and therefore further changes will be needed. On the other hand, if a limited number of developers modify a file, they have a better control of the file changes, and thus unintended changes would be limited. The null hypothesis being tested is H_{02}: *the change entropy of files does not significantly increase with the number of developers that have changed the file*.
- **RQ3:** *How does the change entropy vary between classes participating or not to design patterns?* This research question investigates whether the change entropy is correlated with participation of classes in design patterns. The rationale here is that, on the one hand design patterns could possibly be an indication of good design, which could possibly affect the change entropy. On the other hand, classes participating in certain kinds of design patterns are involved in core features of the application—as shown in previous work (Aversano et al. 2007)—and thus they could change more, and have a higher change entropy. For this research question we only report results for the two Java systems, for which it has been possible to detect design patterns using a tool developed by Tsantalis et al. (2006). Table 2 reports the number of change sets affecting classes participating (and not) to different kinds of design patterns for ArgoUML and Eclipse-JDT. The null hypothesis being tested is H_{03}: *there is no significant difference in the change entropy of classes participating and not to design patterns*.
- **RQ4:** *How does the change entropy relate to the different topics of changes occurred in the system?* This research question investigates whether different kinds of changes—identified by the terms used to comment a change in the CVS/SVN commit note—exhibit a significantly different change entropy. The

Table 2 Number of change sets involving (and not) detected design patterns in ArgoUML and Eclipse-JDT (*AC*: Adapter-Command, *C*: Composite, *D*: Decorator, *FM*: Factory Method, *O*: Observer, *P*: Prototype, *PR*: Proxy, *S*: Singleton, *SS*: State-Strategy, *TM*: Template Method, *V*: Visitor)

System	No	Design pattern										
	pattern	AC	C	D	FM	O	P	PR	S	SS	TM	V
ArgoUML	3,112	1,309	6	74	117	93	384	49	939	1,048	794	0
Eclipse-JDT	5,769	1,191	23	278	189	193	186	239	292	1,285	578	67

conjecture is that some kinds of changes, such as focused bug fixings, exhibit low change entropy, while the introduction of a new feature tends to exhibit high change entropy because of its larger impact on the system. The null hypothesis being tested is H_{04}: *there is no significant difference in the change entropy measured for different kinds of changes*.

- **RQ5:** *How do the interactions among the factors investigated in RQ1, RQ2, RQ3, and RQ4 influence the change entropy?* This research question investigates whether the four factors addressed in RQ1, RQ2, RQ3, and RQ4 interact. In principle, we should consider all possible combinations of the factors: *refactorings*, *number of committers*, *participation of classes in design patterns*, and *change topics*. However, we have selected a subset of these interactions that we believe are more meaningful:
  - the interaction between refactorings and number of committers. It is worth noting that we are not claiming causality; the conjecture is that either the effect of refactoring could depend on the effect of the number of developer, or the other way around. The null hypothesis being tested is H_{05rc}: *there is no significant interaction between the presence of refactorings and the number of committers that modified a file until the refactoring occurred, with respect to the observed change entropy variation*.
  - the interaction between refactorings and the participation of classes to design patterns. The conjecture is that, with respect to change entropy, refactoring can be more or less beneficial to classes participating (or not) in certain design patterns. The null hypothesis being tested is H_{05rp}: *there is no significant interaction between the participation of classes in design patterns and refactoring activities, with respect to the observed change entropy variation*.
  - the interaction between participation of classes in different kinds of design patterns (or to no design pattern) and the number of committers modifying the class file, with respect to the change entropy variation. The null hypothesis being tested is H_{05cp}: *there is no significant interaction between the participation of a class in certain kinds of design patterns and the number of committers modifying the class file*.
  - the interaction between the number of committers modifying the file and the topics describing the change. The conjecture here is that the effect of particular kinds of changes and the file change entropy is related to how many committers modified the file. The null hypothesis being tested is H_{05ct}: *there is no significant interaction between the topic describing the change and the number of committers that modified a file until the change occurred, with respect to the observed change entropy*.

We do not consider the interaction between refactorings and change topics, since we detect refactorings from commit notes, i.e., refactorings will be associated to certain topics identified from such commit notes.

3.2 Variable Selection

This section describes the dependent and independent variables used in the study. Section 3.3 describes how they are computed. The independent variables are:

- *the kind of change* occurred to a source code file in a commit, i.e., (i) refactorings or (ii) other kinds of changes;
- *the number of committers* that modified a file up to a given period when the change entropy is estimated;
- whether a change involves a class participating to one or more *design patterns*. We consider that the class participating to the design pattern was changed if the file where the class is defined was changed (though this could lead to cases of imprecision due to multiple classes declared in the same file, this did not occur for classes participating in ArgoUML and Eclipse-JDT design patterns). If a class participates in more design patterns, then the change is counted for all design patterns in which the class participates.
- *the topics* mentioned in the CVS/SVN commit notes.

The dependent variables are:

- *the normalized, line based change entropy* of a change, computed according to (3).
- *the change entropy variation* for a file before and after a refactoring. Given a file f_i, and the change sets, s_1, s_{m_i}, where f_i was changed, the change entropy variation is computed as follows:

  1. let $fe_{i_1}, fe_{i_2}, \ldots fe_{i_{m_i}}$ be the change entropy of f_i, computed according to (3), for all change sets where f_i was changed;
  2. let s_j be a change set where a refactoring activity was performed;
  3. we compute the average file change entropy of f_i for all change sets preceding s_j, up to the previous refactoring before s_j involving f_i or, if there was no other refactoring before, up to the first change set where f_i was changed. We call this value *change entropy before refactoring*;
  4. similarly, we compute the average file change entropy of f_i for all change sets following s_j, until the following refactoring after s_j involving f_i or, if there was no other refactoring after, until the end of the observed period. We call this value *change entropy after refactoring* in s_j;
  5. finally, the *change entropy variation* is computed as *change entropy after refactoring—change entropy before refactoring* in s_j.

  For example, if we look at Fig. 2, a refactoring occurred in change set s_4 and then again in s_9, thus the change entropy variation related to s_4 is computed as:

$$1/4 \cdot (H(s_5) + H(s_6) + H(s_7) + H(s_8)) - 1/3 \cdot (H(s_1) + H(s_2) + H(s_3))$$

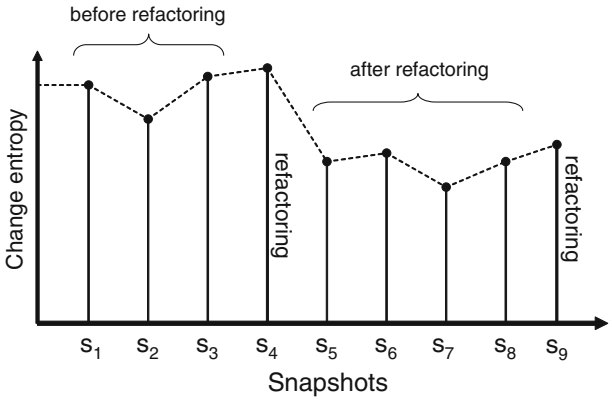


Fig. 2 Variation of change entropy before and after a refactoring

i.e., the difference between the average change entropy of change sets after the refactoring and the average change entropy of the change sets before the refactoring. It is worthwhile to note that the change entropy of the refactoring change itself (i.e., of s_4) is not considered. In principle, the refactoring itself could exhibit a high change entropy, as it is composed of many changes affecting several files; however, we are interested to see if, *after* such a refactoring, the entropy of future changes results affected. In summary, the change entropy variation has the purpose of indicating whether, after a refactoring, changes to a file exhibit a smaller entropy than previous changes and, thus, if the refactoring has reduced the file change complexity.

3.3 Data Extraction Process

The data extraction process comprises six steps, described below.

Step 1: Extraction of Change Sets from Versioning Systems The first step aims at extracting change information from the CVS/SVN log, specifically: (i) the changed files and their revision number (or the repository version for SVN); (ii) the date when a change occurred; (iii) the committer identifier; and (iv) the commit note. After that, we cluster together logically related changes. For this purpose, we rely on a technique that considers the evolution of a software system as a sequence of *Snapshots* (s_1, s_2,...,s_m) generated by a sequence of *Change Sets*, representing the changes performed by a developer in terms of added, deleted, and changed source code lines (Gall et al. 2003). For CVS repositories, as mentioned in Section 2, we identify change sets based on a time-windowing approach (Zimmermann et al. 2004). For SVN repositories (ArgoUML in our study) we use the SVN change sets. Also, it is worthwhile to note that, for the Java systems, we consider only Java files (and not all other files in the repository), and, similarly, for C/C++ files we considered C/C++ source files and header files only. Thus, change sets not considering source files have been discarded.

Step 2: Computing the Entropy of Changes The second step aims at computing: (i) the normalized line-based entropy of a change (3), and (ii) the change entropy of each file touched in a change set, weighted by the number of affected lines (5).

Hassan (2009) proposed to compute the change entropy over different intervals, namely time spans, as for example two weeks, a given number of commits, or a change set. We were interested to analyze the entropy of a specific change, as well as to compare the entropy of different changes, for example changes before and after refactorings, or changes described according to different topics. For this reason, we computed the change entropy considering all changes occurring in a change set (snapshot), identified using the approach of Zimmermann et al. (2004).

In our previous paper (Canfora et al. 2010) we computed the change entropy for a fixed number of commits (500) instead than considering change sets. On the one hand, larger windows of commits allow to observe the change entropy increase/decrease over a longer period—for example, the change entropy might not immediately decrease after a refactoring, e.g., because a complex change was performed right after it. On the other hand, change sets cluster together related changes only; however, a change set might span over a limited number of commits and thus might not allow to observe the change entropy increase/decrease. To overcome such a limitation, while in Canfora et al. (2010) we observed the difference of change entropy related to the (fixed) commit windows before and after the refactorings, in this paper we consider, as explained in Section 3.2, all changes before and after a refactoring, until another refactoring is found.

Step 3: Identifying Refactoring-Related Changes To address **RQ1**, we are interested to investigate whether refactorings contribute to decrease (or to increase) the entropy of changes occurring after the refactoring. It is important to note that, in this context, we are not strictly interested to object-oriented refactoring (Fowler et al. 1999), but rather to any kind of code re-organization or clean up, including, for instance, removal of deprecated code, warnings etc. Ideally, to identify refactorings, it would be appropriate to analyze a source code change, and determine if such a change is related or not to refactoring. For this study, we used a lightweight approach—proposed by Ratzinger et al. (2008)—aimed at identifying refactorings by inspecting CVS/SVN commit notes and mining keywords likely describing refactoring activities, e.g., "refactoring" or "cleanup" or "removed unneeded". This analysis identified 636 candidate refactoring change sets for ArgoUML, 338 for Eclipse-JDT, 435 for Mozilla, and 438 for Samba.

The approach for identifying refactorings described above does not guarantee to identify all refactorings, e.g., it misses refactorings not mentioned in the commit notes. However, it allows for identifying a set of commits that, according to what declared by the developers, are related to refactorings. This is enough to address **RQ1** on a sample of refactorings.

While it would not be feasible to deal with false negatives—i.e., one should know what refactorings were performed in the analyzed systems—we can, at least, make a claim based on results obtained on such a (verified) sample of refactorings. This has been done in two different ways.

The first assessment consists of a manual analysis of all commit notes, to check whether the classification into refactoring was meaningful. This was done by one of

the authors, and another author double-checked the classification, to minimize the presence of false positives.

The second assessment aimed at determining—on a sample of 50 randomly selected change sets for each system classified as refactorings—whether the change in the source code corresponds, or not, to a refactoring. This was done by manually inspecting the file differences—computed with the Unix *diff*—of these change sets. The sample size is not large, but we keep this limited considering the manual analysis of the source code change. Nevertheless, for data sets having a distribution of refactorings not known a priori, for a significance level of 95 %, and for a population size up to 636 (which is the highest number of candidate refactorings we found, i.e., for ArgoUML), the chosen sample size ensures a confidence interval below 10 % (Sheskin 2007).

The inspection produced the following outcome:

- for ArgoUML, there were 13 cases (26 %) in the sample, with commits labeled as "Style issues" concerning code indentation/formatting;
- for Eclipse-JDT, there were 7 cases (14 %) related to JavaDoc improvements. To some extent, this is a case of code re-documentation, that could potentially help to reduce the change entropy;
- for Mozilla, there were 5 (10 %) cases of false positives, actually related to bug fixings;
- for Samba, 4 cases (8 %), related to removing variables unused in makefiles in preprocessor directives were considered as potential false positives, though one can argue that they are specific cases of dead code removal.

In summary, the found "potential" false positives were considered as acceptable cases, and also are a small percentage of the inspected sample. The only exception is represented by the ArgoUML "Style issues". To remove such cases from the ArgoUML dataset, we performed a further manual analysis on all (331) change sets labeled as "Style issues", and removed 106 candidate refactorings from the initial set (that was of 636 candidate refactorings, thus the final set is of 530 refactorings as shown in Table 1).

Step 4: Counting Committers This step aims at measuring the number of committers that have changed a source code file since its introduction in the repository up to a given change. This will be used to address **RQ2**, and thus to relate the change entropy—measured in a given period—with the number of committers that modified the file up to that period.

We are aware that the number of committers might not reflect the number of developers who actually modified a source code file, as only a subset of them could have access to the CVS/SVN repository (Di Penta and Germán 2009). However, for this study, we focus on committers rather than on authors, as we expect that in cases where a limited number of committers correspond to a relatively larger number of authors, the committers could somewhat act as coordinators of the multiple authors and thus limit the file change entropy in the CVS/SVN repository. In essence, considering committers rather than developers would provide a lower-bound of the relation between change entropy and number of developers. Future work will aim at investigating whether change entropy is also influenced by the total number of authors for a file.

Step 5: Identifying Design Pattern Classes and Their Changes The identification of design patterns on each release of the system is performed using a graph-matching based approach proposed by Tsantalis et al. (2006), with the same detection process (and data) used in our previous work (Aversano et al. 2007, 2009).

The Tsantalis et al. approach is based on similarity scoring between graph vertices. It takes as inputs both the system and the pattern graph (cliché) and computes similarity scores between their vertices. Due to the nature of the underlying graph algorithm, this approach is able to recognize not only patterns in their basic form—the ones perfectly matching the cliché described in the Gamma et al. book (Gamma et al. 1995)—but also modified versions (variants). The analysis has been performed using the tool[5] developed by Tsantalis et al., which analyzes Java bytecode. The tool identifies the two main participants (i.e., super-classes) of each pattern, and is able to detect the following patterns: Object Adapter-Command, Composite, Decorator, Factory Method, Observer, Prototype, Proxy, Singleton, State-Strategy, Template Method, and Visitor. The whole set of classes belonging to a pattern is made of main participants and their descendants, that we identify using an analyzer we had built using JavaCC.[6]

Once identified the instances of design patterns in each release of the software system, it is needed to trace them, i.e., to identify whether a pattern instance identified in release j represents an evolution of a design pattern instance identified in release $j-1$. This allows for reconstructing the history of each pattern instance, i.e., in which release it was created and when it was removed. To build the pattern history, we assume that a pattern instance in release j represents the evolution of a pattern instance in release $j-1$ if and only if (i) the type of pattern is the same; and (ii) at least one of the two main participant classes in the pattern is the same class in both releases $j-1$ and j.

For ArgoUML, we analyzed 18 releases in the time interval considered, between release 0.10, and 0.28.1. For Eclipse-JDT, we analyzed 16 releases between 1.0 and 3.6.1.

After patterns belonging to different releases have been traced, we identify, from the change information of *Step 1*, the set of commits in which a class participating in a design pattern undergo changes.

Step 6: Identifying Change Topics To address **RQ3**, we use the Latent Dirichlet Allocation (LDA) (Blei et al. 2003) to identify which topics are described in a commit note. LDA allows to fit a generative probabilistic model from the term occurrences in a corpus of documents. In our case the set of documents is composed of the set of commit notes extracted from the system under analysis. The fitted model is able to capture an additional layer of latent variables which are referred as topics. As the number of topics has to be fixed a priori, we determine it with a data-driven approach. In other words, we start by fixing a large number of topics (30 in our case), and reduce it if we find duplicate topics, i.e., topics described by the same set of words, until no more duplicate topics are found. At the end we determined that the analyzed systems could be described by a set of 10 topics.

[5] http://java.uom.gr/~nikos/pattern-detection.html

[6] https://javacc.dev.java.net

Table 3 Quality of commit notes

System	Total	Empty	(%)	All words			Without stop words		
				1Q	Median	3Q	1Q	Median	3Q
ArgoUML	4,040	99	(2.45 %)	5	9	15	4	6	9
Eclipse-JDT	6,687	806	(12.05 %)	3	5	9	2	4	6
Mozilla	31,891	39	(0.12 %)	7	12	19	4	6	10
Samba	2,326	1	(0.04 %)	10	20	41	6	12	23

Then, each commit note is mapped onto the topic to which it is more likely to belong. This is done by estimating the probabilities of each topic for that commit note. The whole topic analysis has been performed using the *topicmodels*[7] package of the R statistical environment.

Before applying LDA, the corpus of commit notes is preprocessed to: (i) extract terms, pruning out all special characters and numbers; (ii) remove stop words, such as, committer ids, programming language keywords, file and directory names; and (iii) stem terms using the Porter's algorithm (van Rijsbergen et al. 1980).

As for the analysis of refactorings, we are aware that also the topic analysis is affected by the quality of the commit notes. Since determining the quality of such commit notes in an objective-way would not feasible, we provide a coarse, though still quantitative, indication of their quality, in terms of (i) the number and percentage of empty commit notes, often related to the repository initialization, (ii) the distribution of number of words in the commit notes, and (iii) the distribution of number of words in the commit notes after removing stop words. Results are reported in Table 3. The percentage of empty commit notes varies between 0.04 % of Samba and 12.05 % of Eclipse-JDT. The median number of words varies between 5 (4 after stop words removal) for Eclipse-JDT (again the worst case) and 20 (12 after stop words removal) for Samba.

3.4 Analysis Method

This section describes the statistical analysis procedures performed to address the research questions (and test the null hypotheses) formulated in Section 3.1. All tests have been performed using the R statistical environment,[8] and assume a significance level of 95 %.

To address **RQ1**, we compare, for each refactoring occurred on each file, the *change entropy before refactoring* and the *change entropy after refactoring*. For this purpose, we use a paired, non-parametric test, namely the Wilcoxon paired test. The test is performed as two-tailed, because we are interested to see whether refactorings alter the file change entropy in both directions, i.e., whether they increase or decrease it. Other than testing the null hypothesis, we estimate the magnitude of the mean change entropy *before* and *after* refactorings. To this aim, we used the Cliff's Delta (or d), a non-parametric effect size measure (Grissom and Kim 2005) for ordinal data, which indicates the magnitude of the effect of the main treatment on the

[7] http://cran.r-project.org/web/packages/topicmodels

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

dependent variables. For dependent samples, the Cliff's Delta it is defined as the probability that a randomly selected member of one sample before has a higher response than a randomly selected member of the second sample after, minus the reverse probability:

$$d = \frac{|\text{before}^i > \text{after}^j| - |\text{after}^j > \text{before}^i|}{|\text{before}|\,|\text{after}|}$$

The effect size is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$ (Grissom and Kim 2005).

For **RQ2**, we graphically analyze, using boxplots, how the entropy increases for changes occurring to files that had, up to when the change occurred, a different number of contributors. In other words, we count the *cumulative* number of different committers that, up to that change, have modified the file. Then, we compute the Spearman rank correlation between the change entropy observed for a file in each change set where the file is modified and the number of committers that modified the file until that change set, considering all files and for all change sets.

For **RQ3**, we use the Mann-Whitney test and Cliff's delta to compare the change entropy for classes participating or not to design patterns. Then, we use the Kruskal-Wallis test—which is a non-parametric test for multiple-mean comparison—to check whether different kinds of design patterns exhibit different change entropy. After, we perform a pairwise comparison between different design patterns using multiple Mann–Whitney tests, correcting the p-values (since multiple tests are performed) using the Holm's correction (Holm 1979), which sorts the p-values resulting from n tests in ascending order, multiplying the smallest by n, the next by $n - 1$, and so on.

For **RQ4**, after having classified change sets according to their topics, we compare the entropy of changes belonging to different topics using the Kruskal–Wallis test and, again, we use multiple Mann–Whitney tests and Holm's correction to determine which topic exhibits the highest and lowest change entropy.

RQ5 aims at analyzing the interaction among different factors affecting the change entropy or inducing different entropy variations. For H_{05rc} and H_{05rp} we use Kruskal-Wallis test to determine whether files changed by different number of committers, or classes participating in different kinds of design patterns exhibit a significantly different change entropy variation after refactorings. Also, we show boxplots of such variations, to aid understanding whether, in different cases, the variation tends to be positive (increased change entropy) or negative (reduced change entropy). For H_{05cp} and H_{05ct} we use two-way ANOVA to analyze the presence of interaction—with respect to change entropy—between the number of committers modifying a file and (i) the participation of classes in design patterns or (ii) change topics.

4 Results

This section reports results of our empirical study to answer the research questions formulated in Section 3.1. Data for verification/replication are available on-line.[9]

[9] http://www.rcost.unisannio.it/mdipenta/emse-icpc/entropy-rawdata.tgz

4.1 RQ1: How Do Refactorings Affect the Change Entropy?

Figure 3 reports boxplots of the mean change entropy before and after refactorings. With the exception of Samba, after refactorings the change entropy always decreases. Table 4 reports descriptive statistics of the change entropy, together with results of the Wilcoxon test and Cliff's delta effect size. The difference is positive (i.e., the change entropy always decreases) and statistically significant in all cases. The reported effect size indicates that the effect varies in the four systems, i.e., it is small for Eclipse-JDT, negligible for ArgoUML and Samba, while it is large for Mozilla.

Table 5 shows examples of commit notes related to refactoring activities, taken among the most frequent ones. ArgoUML refactoring notes are very informative, and are related to:

1. making resources private, thus increasing the information hiding;
2. re-organizing package interfaces;
3. more generally, re-conceiving part of the architecture, by replacing old structures with new, better designed, ones.

Eclipse-JDT refactoring notes provide less information (as also suggested by statistics in Table 3), but, again, appear to be related to interface cleanup, improvement of information hiding, and code re-organization. Mozilla refactoring notes are informative as well, and belong to categories similar to the two other systems, i.e.,

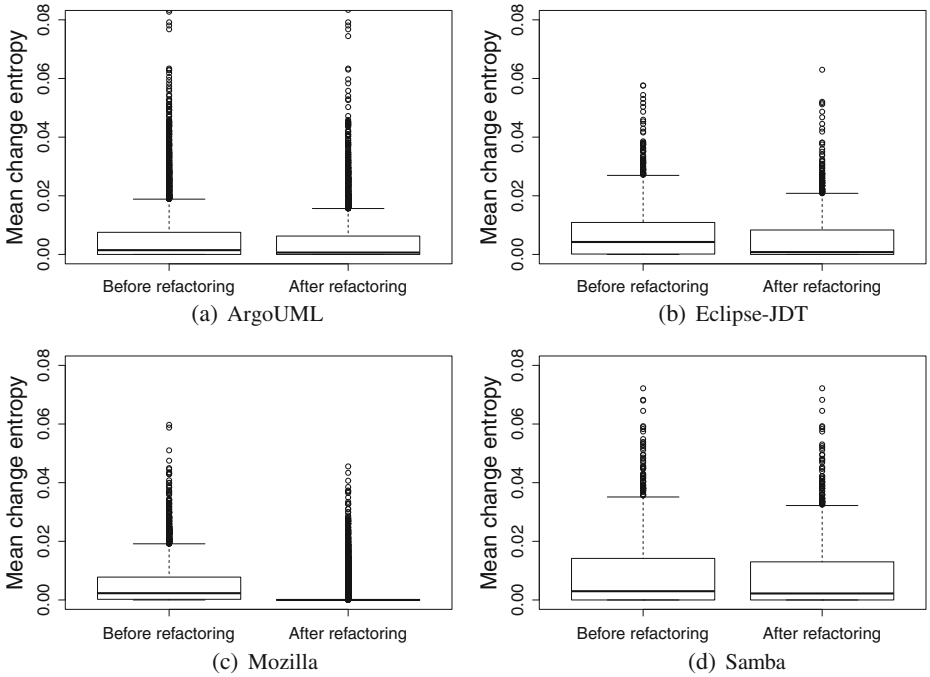


Fig. 3 Boxplots of mean change entropy before and after refactorings

Table 4 Mean change entropy before and after refactorings

Project	Before refactoring			After refactoring			Wilcox test	Eff. size
	Mean	Median	σ	Mean	Median	σ	p value	(d)
ArgoUML	0.01	0.00	0.01	0.00	0.00	0.01	<0.01	0.08
Eclipse-JDT	0.01	0.00	0.01	0.00	0.00	0.01	<0.01	0.18
Mozilla	0.01	0.00	0.01	0.00	0.00	0.00	<0.01	0.53
Samba	0.01	0.00	0.01	0.01	0.00	0.01	0.03	0.04

refactorings, cleanups, removal of unused files and dead code, and renaming of variables. We noticed in this case that refactoring activities were marked with a bug ID and thus also reported in Bugzilla. Samba refactorings vary from the very generic *"Clean up a bit"* or generic refactorings *"More open_pipe_creds() refactoring"* to renamings, specific cleaning up of unused variables, and removal of dead code.

Possible reasons why for Samba differences are not statistically significants are (i) the system is smaller than the others, thus it benefits less of refactorings, and (ii) it is

Table 5 Examples of commit notes for refactoring activities

ArgoUML
- Style issues. Privatised stuff
- Refactoring of the Model component Issue 2696. This part is making the Factory and Helper interfaces accessed from the Model class. . . .
- Replaced deprecated log4j Category with Logger
- Some cleanups in the use of the Model interfaces
- Created a getter for the buttonPanel used it everywhere and deprecated it in favour of addButton(). Consequence of all this: "buttonPanel" is finally private!

Eclipse-JDT
- Refactoring - created new internal package structure
- moved codemanipulation & JavaModelUtil to corext
- API cleanup
- Removed unncessary receivers
- removed unneeded local vars & reduced synthetic accessors

Mozilla
- Bug #141239. Remove tabs from code and clean up indenting in the c files. Not part of the build
- Bug 370185: clean up view header includes (trivial)
- Bug 342311 - xpcom/proxy refactoring in preparation for xptcall rework
- Bug 348748 - Cleanup from the handful of patches which have landed since the initial landing that have readded cast macros
- Bug 330305. Rename nsSelection to nsFrameSelection and deCOMtaminate it removing nsIFrameSelection

Samba
- Clean up a bit
- Cleaned up unused variables returns from non-void functions etc.
- More open_pipe_creds() refactoring
- Moved and renamed DFS error constants from include/rpc_dfs.h to doserr.h tofit in with new error reporting subsystem
- Remove dead function

the only system completely procedural (ArgoUML and Eclipse-JDT are developed in Java, while Mozilla is mostly C++).

In summary, we reject H_{01} and conclude that *the file change entropy decreases after refactoring activities is statistically significant*.

4.2 RQ2: How Does the Change Entropy Relate to the Number of Contributors to a File?

Figure 4 reports histograms of number of changes performed on files that were modified, until that change occurred, by a given number of different committers (x-axis of the histogram). The number of different committers varies between 1 and 19 for ArgoUML, 14 for Eclipse-JDT, 81 for Mozilla, and 17 for Samba.

Figure 5 shows how the file change entropy varies when the number of committers for a file increases. The figure shows boxplots representing the distribution of change entropy related to files that have been changed by a given number of committers until the change when the entropy has been computed. To help evaluating whether the change entropy increases with the number of committers, the graph also reports a (thin, blue) line connecting the medians and a (thick, red) regression line. The change entropy increase is more visible for the Java systems (ArgoUML

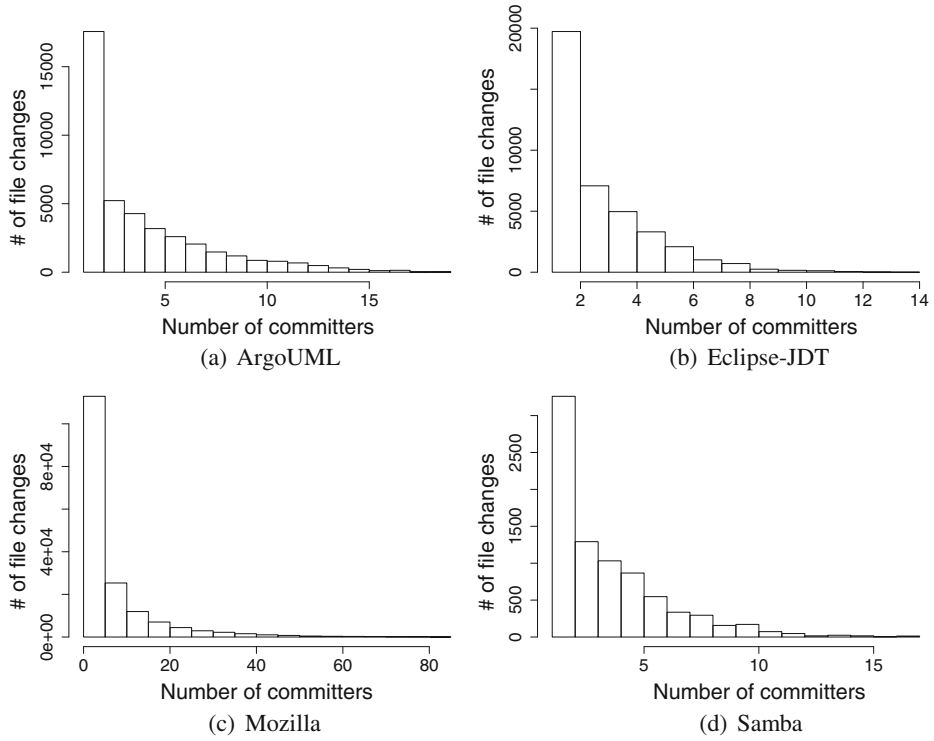


Fig. 4 Histograms of number of commits per files modified by a certain number of committers

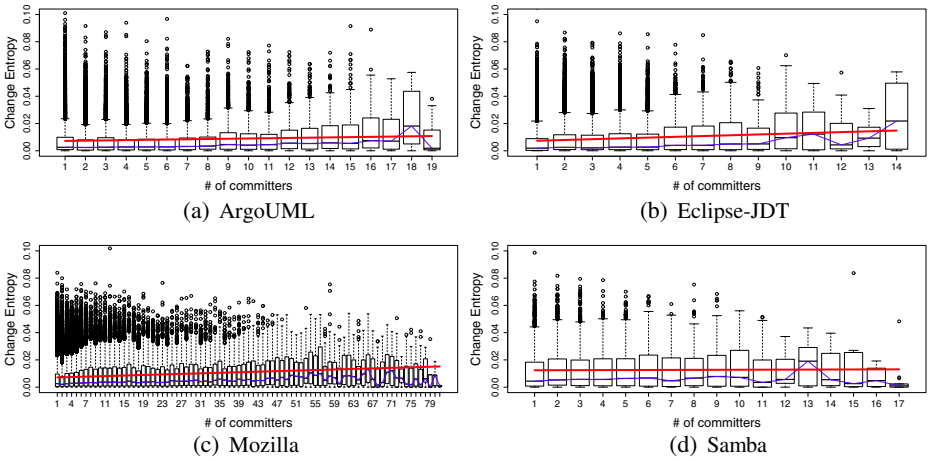


Fig. 5 Change entropy on files modified by a varying number of committers

and Eclipse-JDT) and less visible for the C/C++ systems (Mozilla and Samba). Specifically:

- For ArgoUML the change entropy is stable between 1 and 6 committers, then it starts to slightly increase, with a median—and statistically significant, with p-value < 0.01—increase of 60 % between 6 and 17 committers. It has a peak in correspondence of 18 committers (a further median increase of 61 %), then it decreases again.
- For Eclipse-JDT the change entropy has a median—and statistically significant, with p-value < 0.01—increase of 84 % between 1 and 11 committers. It decreases in correspondence of 12 committers, and then it increases again of 80 % between 12 and 14 committers.
- For Mozilla the change entropy slightly increases with the number of committers, and specifically has a median—and statistically significant, with p-value < 0.01—increase of 70 % between 1 and 50 committers. After, there is an increase of 20 % up to 70 committers, although such increase is not statistically significant.
- For Samba the change entropy has a median increase of 84 % between 1 and 9 committers (though such an increase is not statistically significant), then it slightly decreases again, and then has a peak in correspondence of 13 committers (increasing of 82 % between 11 and 13 committers).

We checked whether there was a significant correlation between the number of committers and the file change entropy. Table 6 reports results of the Spearman rank correlation. In all cases but for Samba the correlation is significant though very low.

Table 6 Spearman rank correlation between file change entropy and number of committers who modified the file until the change occurred

Project	ρ	p-value
ArgoUML	0.05	<0.01
Eclipse-JDT	0.09	<0.01
Mozilla	0.08	<0.01
Samba	0.00	0.79

By looking at the files that, overall, had the highest number of committers, we noticed that, for ArgoUML, the class[10] having the highest number of committers is *ProjectBrowser*, with 19 committers. This class manages the user interface for project browsing, thus changes to project information (performed by various committers) affect this class. Nevertheless, changes to this class are performed in the context of very small modifications, maybe to user interfaces; in fact the change entropy in correspondence of 19 committers (see Fig. 5a) is not particularly high. Another class with a high number of committers is *FigNodeModelElement*, with 18 committers. This is a superclass of a hierarchy of classes containing information about UML elements, thus it has been changed (by various people) any time models changed. Clearly, such a class is being changed with other classes having responsibilities in the model, and this explains the change entropy peak. For Eclipse-JDT, the class *JavaCore* and the class *ProblemReporter* have the highest number of committers (14), and also exhibits (see Fig. 5b) a peak of change entropy. *JavaCore* contains core support for Java plug-ins and is often touched together with other classes providing run-time support. *ProblemReporter*, part of the JDT compiler, handles compiler problem reporting, thus it is likely to be touched as the compiler changes (with other compiler classes, explaining the high change entropy). Other classes with a high number of committers (13) are *PreferenceConstants* (defining some constants) and*JavaPlugin* (containing utility methods). Both contain elements i.e., constants and utilities, that are not often changed together with resources using them. This explains a lower change entropy if compared to *JavaCore* and *ProblemReporter*.

For Mozilla, files with the highest number of committers are *nsDocumentViewer.cpp* (81) and *nsPresShell.cpp* (80). Both are responsible of handling the presentation of documents in a browser. A file exhibiting a peak of change entropy (rev. 1.19) and also a high number of committers (66) is *nsHTMLDocument.cpp*, which models a HTML pages in the browser. This files exhibits also a high change entropy, as maintenance and evolution activities—performed by different developers—dealing the page layout and the HTML document handling—impact this class together with other classes.

For Samba, the file with the highest number of committers is *loadparm.c* (17 committers). This file is responsible for handling all Samba configuration parameters. Also in this case, it is clear that several changes—performed by different developers—would have been related to the Samba configuration options. *server.c*, the main module of the Samba server, has been modified by 16 committers. The highest peak of change entropy is however visible (see Fig. 5d) for *smb.h*, the main header file of Samba, which has been modified by 13 committers. In this case, since this is a header file, it is likely that the changes were performed together with other files using the header, thus explaining the change entropy peak.

In summary, results of **RQ2** lead towards a rejection of H_{02} for all systems except Samba, i.e., *the change entropy is significantly higher for files with a higher number of committers*. For Samba, although there is a slight trend of change entropy increment with the number of committers, the null hypothesis cannot be rejected.

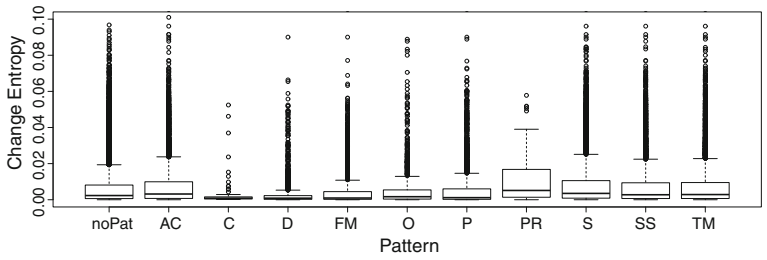
[10]In the following for Eclipse-JDT we refer to classes rather than to files, knowing that in the discussed cases there is a correspondence between a class and a file.

4.3 RQ3: How Does the Change Entropy Vary Between Classes Participating or Not to Design Patterns?

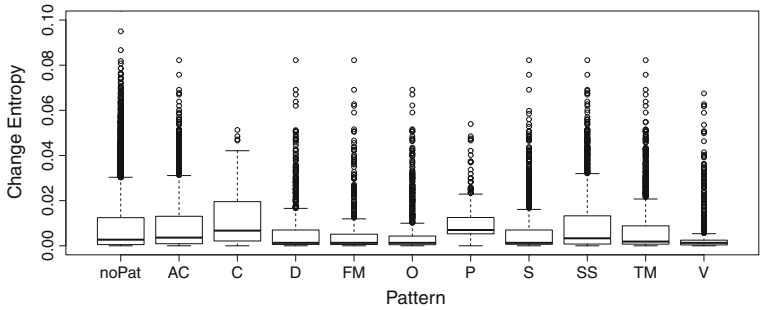
To analyze whether there is a relation between the participation of classes in design patterns and the change entropy, we first analyze whether there is a significant difference between classes participating in at least one design pattern (any kind of design pattern) and classes not participating in any design patterns. Although we obtained a statistically significant difference, the Cliff's delta resulted close to zero. This is because such an analysis is too coarse: different kinds of design patterns might exhibit different characteristics. To this aim, we analyzed the change entropy of each kind of design pattern separately.

Figure 6 shows boxplots of the change entropy for different design patterns (the *noPat* boxplot refers to classes not participating in design patterns). It should be noted that, for Eclipse-JDT, the PR (Proxy) boxplot is not represented as such patterns did not change.

For ArgoUML, the Kruskal-Wallis indicates a significant difference among different patterns (p-value < 0.01). Figure 6a, indicates that Adapter-Command (AC), Singleton (S), State Strategy (SS), Template Method (TM) and Proxy (PR) exhibit a higher change entropy than other patterns, and classes not participating in any design pattern. Table 8 of Appendix reports results of a pairwise comparison between all kinds of patterns. Specifically, the table shows Cliff's delta for cases in which the design pattern indicated on the row exhibits a change entropy



(a) ArgoUML



(b) Eclipse-JDT

Fig. 6 Change entropy for classes participating and not in design patterns

significantly higher than the design pattern on the column. In summary, results indicate that:

- Classes participating in Proxy design patterns have a higher change entropy than classes that do not participate in any design pattern, and than classes participating in Composite, Decorator, Factory Method, and Observer, while no significant difference can be found with Singleton, State Strategy, and Template Method. Proxies are used to implement Combo Boxes containing "proxy" information about models (e.g., class *UMLComboBoxModel*). Clearly, such proxies need to be changed when the models are enhanced, and together with them.
- Classes participating in Composite, Decorator, Observer, Factory Method, and Prototype exhibit a significantly lower change entropy than others. In ArgoUML, there are classes modeling code fragments—e.g., *ClassCodePiece*, *InterfaceCodePiece*, or *PackageCodePiece*—which participate in Composites (as they compose to create models), Decorators (to add features to these fragments), and to Observers (to implement model-view-controllers and thus visualize the fragments). These classes represent specific code fragments, that do not change when models change, while underwent changes almost independently from other artifacts.

For Eclipse-JDT, the Kruskal-Wallis test indicates a significant difference among different patterns (p-value < 0.01). Figure 6b shows that Prototype and Composite exhibit a higher change entropy than other patterns. Pairwise differences, reported in Table 9 of Appendix, indicate that:

- classes participating in Adapter-Command, Composite, State-Strategy, and Prototype have significantly higher change entropy than other classes. Adapters change when there is need for re-adapting a changed interface, thus, very likely, the change involves classes playing the role of Adapter, Adaptee, and other classes related to the Adaptee. Prototypes are used to create Abstract Syntax Tree (AST) classes, and Composites are used to model whole-part relationships in ASTs. State Strategy patterns are used in the Eclipse-JDT compiler and in tools such as the IDE code completion.
- classes participating in Observer and Visitor have a significantly lower change entropy than other classes. In Eclipse-JDT, visitors are used to build analyzers and code generators upon the AST. Many changes involving visitors were very focused, impacting few classes and few lines of code. Some examples were changes related to code completion *"codeassist enhancement for variable initializer and assignment"* which required to modify a single line on three classes. Observers are mainly used for model-view-controller, and by doing a code and CVS log inspection we noticed that some of these classes underwent small changes (2 lines) to *"Reduce API footprint"*.

In summary, we reject H_{03}, stating that *classes participating in different kinds of design patterns exhibit a different change entropy*. However, as we also found in a previous paper where we analyzed the change proneness of classes participating in design patterns (Aversano et al. 2007), the change entropy depends not only

on the kind of design pattern itself, but also on the purpose a design pattern is used for.

4.4 RQ4: How Does the Change Entropy Relate to the Different Topics of Changes Occurred in the System?

Table 7 reports the topic descriptions for the four projects, as well as the number and percentage of change sets mapped onto the different topics.

Table 7 reports the 10 highest ranked words describing the topics detected on the commit notes of the four systems, as well as the number of percentage of change sets related to that topic. Besides change sets with empty commit notes, all others were mapped onto topics. For ArgoUML, the most frequent topics are (i) topic 10 (19 %) which is among other things mainly related to modifying the user interface of the tool; (ii) topic 2 (16 %) with dealt with models (and stereotypes in particular), (iii) topic 5 (15 %) which contains the style issues also discussed in Section 3.3, and (iv) topic 4 (14 %) with is mainly related to refactoring activities. For Eclipse-JDT, we have (i) topic 1 (19 %) related to bug fixings, (ii) topic 10, mainly related to licensing (i.e., changes related to moving from Common Public License to Eclipse Public License, as described in a paper by Di Penta et al. (2010)). For Mozilla, (i) topic 5 (22 %) is related to page visualization/reflowing issues, and (ii) topic 6 (19 %) is related to fixings related to browser crashes. For Samba the most frequent topics are topic 8 (13 %) and topic 7 (12 %) related to different kinds of bug fixings.

Figure 7 reports boxplots of change entropy for change sets related to different topics. Results of the pairwise analysis are reported in Tables 10, 11, 12, and 13 of Appendix. The Kruskal-Wallis test always indicated a significant difference of change entropy (p-value < 0.01 in all cases) among topics.

For ArgoUML, topic 9 has a change entropy significantly higher than other topics, and, among the most frequent topics, also topics 3 and 4 exhibit a high change entropy. This result is surprising, at least for topics 4 and 9, as these topics are largely related to refactorings. However, this does not contradict findings of **RQ1**: while results of **RQ1** indicate that the change entropy decreases *after* a refactoring, as shown here a change related to refactoring might likely involve many files and source code lines. This is also confirmed in Eclipse-JDT, where topic 6, related to refactoring, exhibits a higher change entropy than other topics. Topic 3 and 4 have a lower change entropy than others: they are related to licensing changes and updates to copyright years. While this can involve many files, the change is so limited (it should be a number in a year or a letter in a word, i.e., the word *cpl*, Common Public License, became *epl* (Eclipse Public License) (Di Penta et al. 2010)), that the change entropy is very low. For Mozilla, topics with the highest change entropy are topic 6 (crashes), topic 7 (which involved changed to API), topic 9, and topic 10 (refactorings, as in the previous two projects). The change entropy is lower for topic 8 (licensing change/update) than for other topics. For Samba, results do not really show significant differences among topics. Only topic 9 has a (slightly) lower change entropy than other topics, as it is related to copyright updates as in Eclipse-JDT.

We conclude by rejecting H_{03}, i.e., *changes classified into different topics exhibit a significantly different entropy*.

Table 7 Topics identified in the commit notes of the four projects using LDA

#	# Change sets (%)	Top ranked words
ArgoUML		
1	79 (2 %)	profil, intern, work, branch, merg, issu, trunk, app, initi, head
2	641 (16 %)	issu, fix, work, remov, implement, fig, stereotyp, improv, object, set
3	191 (5 %)	fix, line, code, modelfacad, access, convert, issu, mostli, list, action
4	548 (14 %)	remov, deprec, method, chang, call, nsuml, code, refactor, file, function
5	596 (15 %)	issu, style, fix, first, indent, stuff, error, file, pass, same
6	213 (5 %)	fix, make, hide, compon, nsuml, bug, part, factori, refactor, will
7	518 (13 %)	javadoc, remov, issu, fix, comment, warn, updat, nsuml, patch, few
8	135 (4 %)	copyright, updat, chang, make, year, statement, sourc, open, forgotten, move
9	257 (7 %)	replac, improv, someth, modelfacad, mthing, instanc, clean, singl, comment, nsuml
10	756 (19 %)	issu, number, remov, depend, add, chang, panel, properti, delet, proppanel
Eclipse-JDT		
1	1085 (19 %)	bug, fix, move, file, assist, method, properti, quick, breakpoint, content
2	720 (13 %)	chang, fix, bug, keyword, substitut, format, work, string, set, explor
3	191 (3 %)	copyright, updat, head, fix, autom, date, epl, organ, notic, replac
4	294 (5 %)	tool, bug, autom, releng, copyright, context, initi, help, delet, featur
5	700 (12 %)	type, path, build, progress, extern, page, sourc, jar, gener, folder
6	484 (8 %)	bug, miss, exist, remov, review, api, error, deprec, updat, adapt
7	642 (11 %)	work, bug, clean, set, check, tag, non, fix, project, type
8	489 (9 %)	api, bug, improv, renam, convers, evalu, access, fix, packag, stori
9	314 (5 %)	remov, trail, clean, add, field, import, pass, head, space, whitespac
10	850 (15 %)	head, cpl, epl, bug, convert, merg, back, ing, renam, stream
Mozilla		
1	2609 (8 %)	remov, warn, chang, defin, make, method, compil, checkin, initi, iid
2	3827 (12 %)	remov, move, need, folder, longer, code, clean, part, asa, copi
3	1312 (4 %)	class, moz, spec, chang, wrapper, impl, code, method, interfac, implement
4	3862 (12 %)	branch, land, back, initi, chang, patch, code, bustag, remov, fix
5	6957 (22 %)	frame, chang, make, fix, reflow, set, code, handl, implement, first
6	6063 (19 %)	fix, patch, problem, leak, crash, chang, window, make, code, part
7	1818 (6 %)	chang, updat, api, licens, chofmann, boilerpl, xpl, endico, interfac, code
8	605 (2 %)	chang, licens, gpl, lgpl, mpl, tri, take, relicens, round, most
9	1796 (6 %)	chang, implement, remov, method, makefil, alloc, api, refer, addref, interfac
10	2961 (9 %)	remov, part, code, unus, directori, work, make, done, patch, goodbyeeee
Samba		
1	267 (11 %)	fix, code, remov, name, patch, com, global, compil, check, chang
2	242 (10 %)	fix, chang, compil, gcc, macro, debug, memori, pedant, isxxx, call
3	206 (9 %)	chang, code, user, call, work, patch, fix, command, databas, mode
4	321 (14 %)	code, fix, merg, function, chang, head, move, you, safe, printer
5	221 (10 %)	file, function, string, code, initi, make, need, just, call, branch
6	153 (7 %)	rpc, cli, branch, pars, code, handl, set, same, pipe, srv
7	283 (12 %)	fix, name, code, user, head, function, chang, sync, lookup, call
8	303 (13 %)	fix, code, chang, name, kei, function, work, need, make, add
9	134 (6 %)	updat, copyright, made, code, subnet, chang, work, winbind, master, add
10	191 (8 %)	chang, remov, file, name, header, add, fix, subnet, netbio, trust

4.5 RQ5: How Do the Interactions Among the Factors Investigated in RQ1, RQ2, RQ3, and RQ4 Influence the Change Entropy?

Finally, we analyze whether there is an interaction among the different factors (related to the change entropy) we considered.

First, we analyze whether there is a significant interaction—with respect to the dependent variable change entropy variation—between the number of committers that changed a file and refactoring activities. In other words, we observe whether the change entropy variations after a refactoring is significantly different for files changed by a varying number of committers. Kruskal-Wallis test indicates a significant difference (p-value < 0.01). To better understand such differences, Fig. 8 shows boxplots of change entropy variation after refactorings for files modified by a varying number of committers. The (red) horizontal line represent a median of zero variation. For systems with a relatively small number of committers modifying a file (Eclipse-JDT, and Samba), the boxplot median tends to stay below zero, i.e., the refactoring reduces the change entropy regardless of the number of committers modifying the file. For ArgoUML the median remains stable around zero. For higher numbers of committers—e.g., 18 for ArgoUML, 12 for Eclipse-JDT, 16 for Samba we can notice how the median tends to stay above zero. For Mozilla, where the number of committers modifying a file is higher, we can clearly notice how the median of change entropy variation increases with the number of committers, reaching and going above zero for a number of committers greater than 50. In conclusion, we can reject H_{05rc} i.e., *the effect of refactoring on change entropy variation decreases when the number of committers changing a file increases.*

We perform a similar analysis to investigate the interaction between refactorings and participation of classes in design patterns with respect to change entropy variations. Also in this case Kruskal–Wallis test indicates a significant difference among

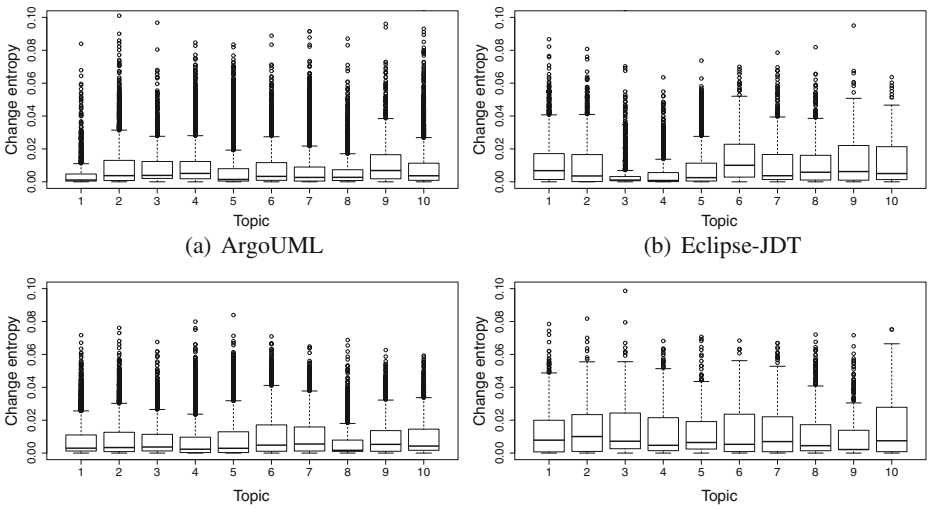
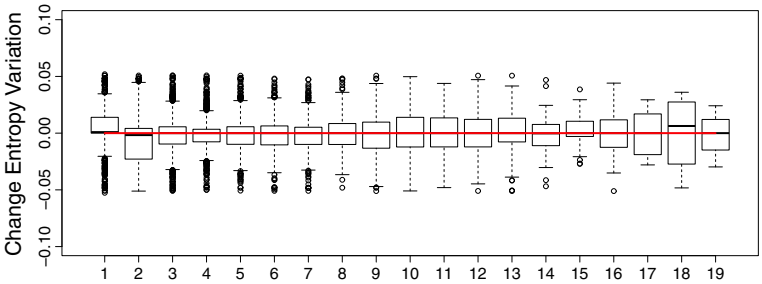
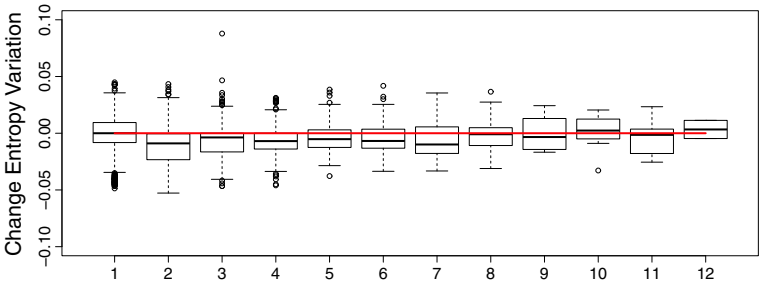


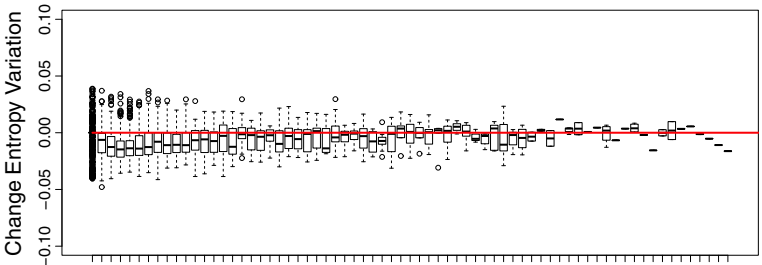
Fig. 7 Entropy of changes belonging to different topics



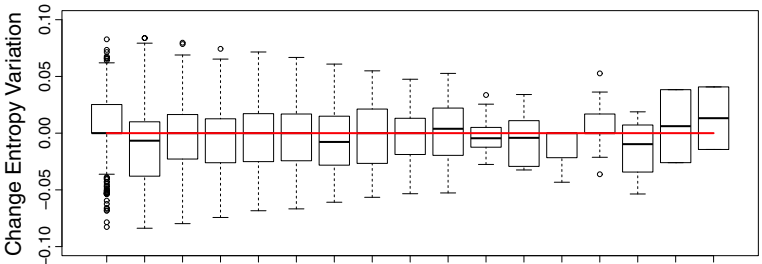
(a) ArgoUML



(b) Eclipse-JDT



(c) Mozilla



(d) Samba

Fig. 8 Boxplots of change entropy variation after refactorings for files modified by a varying number of committers

patterns (p-value < 0.01). Figure 9 shows boxplots of change entropy variation after refactorings for classes participating in different kinds of design patterns (as well as classes not participating in design patterns). For ArgoUML, we cannot really see a difference among patterns, with the exception of Proxy. As mentioned for **RQ3**, Proxy is used to provide surrogate representations of code snippets and models in the ArgoUML visualizations. Refactorings of these Proxy instances helped to limit future changes to fewer classes only, i.e., view or model. For Eclipse-JDT, several design patterns exhibit a change entropy variation with a median below zero. This is particularly true for Composite, for which a refactoring would help to separate concerns among different components, and thus reduce the change entropy. Such a finding is interesting as—while as shown in Fig. 6b and in Table 9 Composite tend to exhibit a higher change entropy—refactoring on them seem to be very beneficial, in that help to reduce it. Negative variations are also visible for Observer (where refactorings would again improve decoupling) and Prototype (used to create AST elements, therefore refactoring reduced the entropy of changes upon AST nodes). Thus, we can conclude by rejecting H_{05rp} by stating that *at least in some cases, refactorings are related to different change entropy variations for classes participating to different design patterns.*

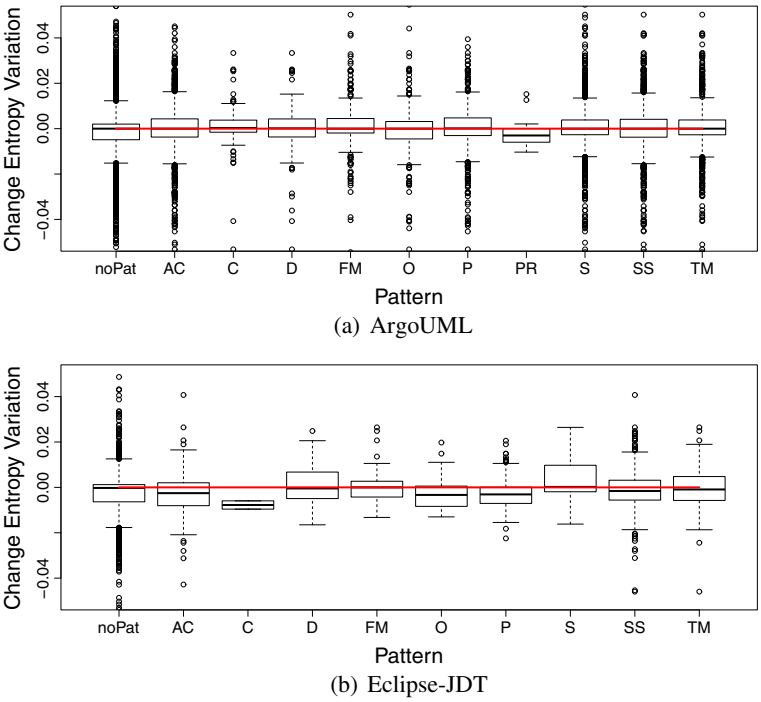


Fig. 9 Boxplots of change entropy variation after refactorings for classes participating in different kinds of design patterns

We also analyze whether there is an interaction between the participation of classes in design patterns and the number of committers modifying the class file, with respect to the file change entropy. Though it would not be possible to analyze different cases in detail, two-way ANOVA indicates, for both ArgoUML and Eclipse-JDT, the presence of a significant interaction between the two factors. For ArgoUML, a deeper analysis indicates that classes participating in Proxy patterns have a higher number of committers than others (six, while such a median number is five for Composite and Prototype, and lower than five in other cases). As we can recall from **RQ3**, Proxy classes also exhibit a higher change entropy than other classes. For Eclipse-JDT, Composite classes have been modified by a median number of three committers (such a number is two for Prototype and for classes not participating in patterns, one in all other cases). Also in this case **RQ3** indicated that classes participating in Composite and Prototype exhibit a higher change entropy than other classes. Therefore, we can reject H_{05cp} by stating that *change entropy for classes participating to different kinds of design patterns is influenced by the number of committers modifying the class file.*

Finally, we analyze the interaction between change topics and the number of committers modifying a file. Also in this case, we perform two-way ANOVA. With the exception of Eclipse-JDT where we did not found any significant interaction (p-value $= 0.71$), in all other cases the interaction is statistically significant (p-value < 0.01). We further investigated the Eclipse-JDT case, finding that change sets related to different topics occurred on files modified by a similar median number of committers (two for all topics, except topics 2, 5, 7 and 8 where such a median was one). In conclusion, except for Eclipse-JDT, we can reject H_{05ct} stating that *there is a significant interaction—with respect to change entropy—between change topics and number of committers modifying a file.*

5 Threats to Validity

This section discusses the main threats to the validity of our study.

Construct validity threats concern the relationship between theory and observation. These are mainly due to imprecision in the measurements we performed. While we carefully checked the identified refactorings to avoid false positives—as detailed in Section 3.3—we could not make any claim about the recall of the refactoring identification approach used. Thus, our results will be representative of the identified sample only, and there can be other (non documented) refactorings influencing the change entropy differently from what we found in **RQ1**. Committers might represent, as discussed in Di Penta and Germán (2009) a partial view of who modified a file. Future work will aim at considering all file contributors other than only committers, using the approach proposed in Di Penta and Germán (2009) or relying on versioning systems—such as *git*, that can distinguish the committer from the contributors. Design pattern detection is another source of imprecision and of limited recall. Tsantalis et al. (2006) report a precision of 90 % or above for the systems they analyzed. We are aware that the outcome of the analysis on topics performed in **RQ4** strongly depends on the content of the commit notes. Although we could not thoroughly assess their quality, we report information (Table 3) about the percentage of empty notes and descriptive statistics on note lengths. We are

aware that, in some cases, especially for Eclipse-JDT, the presence of relatively short commit notes could have influenced the results. Finally, we computed the change entropy on change sets, while Hassan (2009) proposed additional ways of computing it (fixed time window, or fixed set of commits). In this paper, we wanted to characterize the change entropy within a unique change activity performed by a committer—captured as a change set (Zimmermann et al. 2004)—and investigate, in case such a change were a refactoring, how this would affected the average change entropy.

Conclusion validity concerns the relationship between the treatment and the outcome. Attention was paid not to violate assumptions made by statistical tests. We mainly used non-parametric tests, plus ANOVA (only for **RQ5**), which is pretty robust to deviation from normality. Also, when multiple data sets have been compared, we complemented Kruskal-Wallis test with multiple pairwise comparisons, correcting p-values using the Holm's correction. In addition to testing the presence of significant differences, wherever this was useful—e.g., in **RQ1**—we also checked the presence of practically relevant differences using the Cliff's delta effect size.

Threats to *internal validity* concern factors that can influence our observations. The study does not claim any cause-effect relationship between the investigated independent variables (refactoring changes, number of committers, participation of classes in design patterns, change topics) and the dependent variable (change entropy and its variation). However, we discussed our results and looked at the system characteristics and source code trying to provide interpretations to our findings. For **RQ1**, we are aware that our observations can be polluted, other than by the fact that we only analyzed documented refactoring, by the likely presence of other kinds of changes in changes documented as refactorings: e.g., one could fix a bug or add a new feature while doing a refactoring. Also, we acknowledge that the classification of changes by topics reported in **RQ4** is based solely on the commit note text and it could not completely characterize the semantics of the change performed. Finally, we are aware that, although we believe that the factors we investigated are relevant factors influencing the change entropy, these are not the only ones, and there may be others factors we have not taken into consideration.

Threats to *external validity* concern the generalization of our findings. We analyzed four systems of different size, having different characteristics, different organizational structure (e.g., number of committers), and developed with different languages. This allowed us to obtain both findings common to all projects, but also findings specific of projects exhibiting some particular characteristics. Having said that, we believe that it is highly desirable to replicate this initial study on further systems.

6 Related Work

This section describes related work concerned with (i) the use of entropy as a measure of disorder in software and (ii) the use of topic models to analyze software system evolution.

The proposal of entropy-based metrics to measure the disorder of a software system is not recent. Chapin (1995) proposed the use of such a metric to suggest

maintenance activities when the entropy begins to grow. Harrison (1992) proposed the entropy as a measure of software complexity. Bianchi et al. (2001) proposed an entropy-based metric to predict and monitor the degradation of a software system. In particular, they showed that software degradation, measured by maintenance effort and number of defects, is correlated with the software entropy.

Eick et al. (2001) modeled code decay using measures different from entropy. Given a change, they considered as indicators of code decay the number of deltas associated with a change, the amount of lines added and removed (which we also consider, see (3), the date when the change is completed, the time needed to implement the change, and the number of developers implemented the change (which, again, is related to the committers factor we considered in our study). Other measures of change complexity and their relation with fault-proneness were proposed by Nagappan and Ball (2007), who used code dependencies and measures from code churns (changed LOC, number of files changed, and number of changes) between two program versions to predict failures.

Only recently, with the advance of techniques used to mine software repositories, that entropy-based metrics are becoming valuable, as they can be estimated with from huge amount of change information stored in version repositories. Hassan and Holt (2003) used the entropy as an indicator of how much information exists in the development process. They pointed out that too much information will require more effort for developers to keep track of the changes over time, thus the higher the entropy of a system, the more complex the system's code becomes over time. In a subsequent work, Hassan (2009) introduced a definition of complexity of code changes based on entropy, and showed that it could be used as a predictor of fault-proneness.

In summary, the works discussed above modeled software disorder/degradation using the concept of entropy, and related it with fault and change-proneness. We share the measures of entropy above defined, in particular the one defined by Hassan (2009) for code changes. However, we show that some kinds of changes, like refactorings (in a broader sense) contribute to decrease the change entropy, and investigate the relationship between the change entropy and other factors, such as the number of committers, the participation of classes in design patterns, and the change topics.

Capiluppi et al. (2007) studied the evolution of an agile project, finding that the complexity of control work, e.g., related to refactoring activities, is higher than other activities. In **RQ4**, we also found that refactoring changes exhibit a high change entropy, although we also found (**RQ1**) that the change entropy decrease after refactoring activities.

Recently, the use of topic models, such as those based on LDA, has attracted attention due to the availability of large amount of text in software repositories. Thomas et al. (2010) adopted topic models to describe software evolution. The study introduces a set of topic evolution metrics and shows how such metrics are correlated with typical change events. The study shows that topic models can be an effective tool for summarizing software change activities. Kuhn et al. (2007) proposed the use of topics—in this case identified ad dominant terms in concepts obtained with Latent Semantic Indexing—to group source artifacts that use similar vocabulary. The authors demonstrated that such groups, called semantic clusters, could reveal the developer's knowledge hidden in identifiers, supporting effectively the comprehension

task in software maintenance. Linstead and Baldi (2009) introduced a metric based on LDA to measure the coherence of a bug report. Such a metric extends previous coherence measures by considering the entropy of the latent topic distributions that underlie the textual information in bug databases. Linstead and Baldi applied the metric on the Gnome Bugzilla database, and found the metric effective for estimating bug report quality.

We share with previous works the idea of summarizing data from software repositories—and specifically from commit notes—using topic models. In addition, we relate these topics to change entropy.

The present paper is an extension of a previous work by some of its authors (Canfora et al. 2010). The previous paper represents a first investigation of factors influencing the source code change entropy, performed on two Java systems (ArgoUML and Eclipse-JDT). The present work extends the previous one, by analyzing two more systems (Mozilla and Samba), and performing a deeper analysis of the relation between change entropy and the nature of changes, by (i) introducing the notion of change entropy variation, and analyzing it in correspondence of refactorings, and (ii) analyzing the relation between change entropy and the topic describing changes in the versioning system commit notes.

7 Lessons Learned and Conclusions

Software systems evolve for various reasons and, as reported by many studies, changes tend to increase software complexity (Lehman 1980; Lehman and Belady 1985) and to favor software structure deterioration, a phenomenon known as software "aging" (Parnas 1994). Several researchers have modeled such phenomenon by adapting the Shannon's entropy definition (Shannon 1948) to model software system complexity (Bianchi et al. 2001; Chapin 1995; Harrison 1992), and more specifically to characterize the entropy of software changes (Hassan 2009).

Starting from Hassan's work (Hassan 2009), we performed an empirical study aimed at investigating the relations between software change entropy and factors that could relate to it, specifically (i) refactoring activities, at least those documented in commit notes; (ii) number of committers changing a source code file; (iii) participation of classes in different kinds of design patterns, detected in the source code using a tool developed by Tsantalis et al. (2006), and (iv) change topics, identified using Latent Dirichlet Allocation (LDA). The study has been conducted on a portion of history of four open source projects, namely ArgoUML, Eclipse-JDT, Mozilla, and Samba.

Results indicated that (i) overall, the four investigated factors are significantly related with change entropy, (ii) there are cases in which the factors also have a significant interaction, and (iii) specific characteristics of the four projects also played a role in how the four factors were related to change entropy. Specifically, the study showed that:

– *Refactoring activities significantly reduce the change entropy:* we observed that, after a refactoring was performed, the average change entropy tends to be significantly lower than before the refactoring. This can be explained because

refactorings would help to reduce the scattering of changes to be done when evolving a software system; as a consequence, the change entropy tends to be reduced. The effect of refactoring tends to be less visible in Java systems, like Eclipse-JDT and ArgoUML, which have a good design already, or in smaller systems like Samba, while it is more visible in systems such as Mozilla, which is large, only partially object-oriented, and has a pretty long history if compared to the other systems.

- *Files changed by many committers have a higher change entropy:* we found that, in general, files being modified by a higher number of committers exhibit a significantly higher change entropy than files modified by a lower number of committers. This happened in all systems, except Samba that is smaller and has a lower number of committers (25 in total) compared to other projects (which have between 37 and 628 committers). When many persons modify a source code file, the code complexity can increase and its understandability can decrease, because different persons write source code using different coding styles/naming conventions. In general, whoever changes the source code file has to understand code previously written/changed by many different persons, and this increases the risk of misunderstanding, and thus bug introduction, making further changes necessary to fix the bugs.
- *Changes involving classes participating to different kinds of design patterns exhibit a significantly different change entropy:* however, this does not depends only on the specific kind of design pattern, as we found different values of change entropy for the same pattern in different systems. In general, when a design pattern is able to decouple concerns, and thus to keep changes limited to certain source code elements, then the change entropy tends to be very low, as for Composite and Decorator in ArgoUML, and for Visitor in Eclipse-JDT. As shown in a previous paper (Aversano et al. 2007), depending on the use of the pattern, as well as on the system architecture, design patterns may or may not be able to make the system resilient to changes.
- *Changes related to different topics exhibit a different entropy:* in all systems, there are change activities that, intrinsically, have a lower impact on the system than others, and therefore exhibit a significantly lower change entropy than other changes. For example, some bug fixings exhibit a low change entropy, and the same happens when people update copyright years or perform specific, focused changes on licensing statements—e.g. changing *cpl* to *epl* in Eclipse-JDT—while the introduction of relevant features, or also the removal of warnings are changes that tend to be more scattered, and thus exhibit a higher entropy. Another relevant finding is that refactoring-related changes exhibit a high entropy, because a refactoring often affect a substantial amount of source code and sometimes several files. However, as explained above, such high-entropy changes have a beneficial effect, i.e., the entropy of changes following refactorings result significantly decreased, as explained above. Knowing the relationship between change topics and entropy has an important consequence: since, as shown by Hassan (2009), high change entropy is correlated with fault-proneness, when performing high-entropy changes it is advisable to perform activities—such as code inspection, testing, or the analysis of change ripple effect—to minimize the introduction of bugs.

– *Factors affecting change entropy can interact:* not only specific factors—i.e., refactoring activities, number of committers, participation of classes in design patterns, and change topics—are significantly related with change entropy, while sometimes they also interact. When the number of committers is very high—as in the case of Mozilla—the effect of refactorings in reducing change entropy is higher for files having a higher number of committers. For systems having a smaller number of committers—such as ArgoUML, Eclipse-JDT, and Samba—such an interaction is less visible. In summary, this would suggest that refactoring would be particularly useful for projects with a high number of committers. Also, we found that the relation between refactorings and change entropy variation is different on classes participating in different design patterns, for example it does not produce significant effects for Singleton classes, while, in Eclipse-JDT, is very beneficial for Composite.

Our findings suggest that software aging is a phenomenon typical of large projects, with large number of committers, and reveal that not all changes affect aging in the same way. They also confirm that aging can be contrasted by specific actions, such as refactoring.

An interesting question raised by this study is whether or not the effect of the number of committers that changed a file on the change entropy (RQ2) depends upon the open source nature of the analyzed project. A conjecture could be that in a corporate setting, where a shared corporate culture exists and guidelines and standards for design and coding are enforced, the negative effects produced by having many developers working a a file could be mitigated. This conjecture will investigated in future work. Further development of this work aims at extending the study on further projects, as well as at studying relations between change entropy and the social structure of the developers' network.

Appendix: Detailed Analyses

Table 8 ArgoUML: comparison of change entropy among different design patterns

Pattern	noPat	AC	C	D	FM	O	P	PR	S	SS	TM
noPat			0.26	0.24	0.13	0.07	0.06				
AC	0.10		0.29	0.29	0.19	0.14	0.13				
C											
D											
FM				0.17							
O				0.25	0.06						
P				0.24		0.01					
PR	0.49		0.86	1.39	0.71	0.54	0.54				
S	0.14	0.04	0.44	0.43	0.26	0.20	0.19			0.05	0.05
SS	0.07		0.39	0.38	0.20	0.14	0.13				
TM	0.08		0.39	0.38	0.21	0.15	0.13				

The table shows the Cliff's delta where differences are significant

Table 9 Eclipse-JDT: comparison of change entropy among different design patterns

Pattern	noPat	AC	C	D	FM	O	P	S	SS	TM	V
noPat				0.16	0.22	0.18		0.13			0.24
AC	0.05			0.19	0.26	0.22		0.17		0.10	0.28
C	0.20			0.33	0.43	0.35		0.29		0.23	0.43
D											0.12
FM											0.04
O											
P	0.17	0.12		0.47	0.69	0.52		0.28	0.11	0.21	0.42
S						0.07					0.15
SS	0.04			0.23	0.34	0.27		0.19			0.27
TM				0.12	0.23	0.16					0.21
V											

The table shows the Cliff's delta where differences are significant

Table 10 ArgoUML: Cliff's delta resulting from the comparison of change entropy for different topics

Topic	1	2	3	4	5	6	7	8	9	10
1										
2	0.34				0.15		0.13	0.15		
3	0.41	0.01			0.17	0.04	0.14	0.17		0.04
4	0.38	0.01			0.16	0.03	0.14	0.17		0.03
5	0.16							0.01		
6	0.33				0.15		0.10	0.13		
7	0.20				0.03					
8	0.16									
9	0.54	0.13	0.11	0.13	0.34	0.16	0.31	0.34		0.14
10	0.32				0.14		0.11	0.15		

Table 11 Eclipse-JDT: Cliff's delta resulting from the comparison of change entropy for different topics

Topic	1	2	3	4	5	6	7	8	9	10
1		0.08	0.34	0.32	0.15					
2			0.29	0.27						
3										
4			0.04							
5			0.31	0.27						
6	0.16	0.26	0.86	0.81	0.38		0.16	0.18	0.09	0.13
7		0.05	0.51	0.47	0.15					
8		0.04	0.52	0.47	0.13					
9		0.14	0.71	0.66	0.25					
10			0.67	0.62	0.20					

Table 12 Mozilla: Cliff's delta resulting from the comparison of change entropy for different topics

Topic	1	2	3	4	5	6	7	8	9	10
1				0.04				0.09		
2			0.00	0.08	0.01			0.12		
3	0.05			0.08	0.01			0.13		
4								0.05		
5	0.04			0.08						
6	0.18	0.12	0.13	0.22	0.14			0.22		
7	0.15	0.10	0.10	0.20	0.11			0.21		
8										
9	0.10	0.05	0.05	0.14	0.06			0.21		
10	0.10	0.05	0.05	0.14	0.06			0.21		

Table 13 Samba: Cliff's delta resulting from the comparison of change entropy for different topics

Topic	1	2	3	4	5	6	7	8	9	10
1										
2									0.20	
3								0.13	0.22	
4									0.15	
5									0.14	
6										
7										
8										
9										
10										

References

Aversano L, Canfora G, Cerulo L, Del Grosso C, Di Penta M (2007) An empirical study on the evolution of design patterns. In: ESEC-FSE '07: proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. ACM Press, New York, pp 385–394

Aversano L, Cerulo L, Di Penta M (2009) The relationship between design patterns defects and crosscutting concern scattering degree: an empirical study. IET Softw 3(5):395–409

Bianchi A, Caivano D, Lanubile F, Visaggio G (2001) Evaluating software degradation through entropy. In: METRICS '01: Proceedings of the 7th international symposium on software metrics. IEEE Computer Society, Washington, DC, p 210

Blei DM, Ng AY, Jordan MI (2003) Latent Dirichlet allocation. J Mach Learn Res 3:993–1022

Canfora G, Cerulo L, Di Penta M, Pacilio F (2010) An exploratory study of factors influencing change entropy. In: The 18th IEEE international conference on program comprehension, ICPC 2010, Braga, Minho, Portugal, 30 June–2 July 2010. IEEE Computer Society, Washington, DC, pp 134–143

Capiluppi A, Fernández-Ramil J, Higman J, Sharp HC, Smith N (2007) An empirical study of the evolution of an agile-developed software system. In: 29th international conference on software engineering (ICSE 2007), Minneapolis, MN, USA, 20–26 May 2007. IEEE Computer Society, Washington, DC, pp 511–518

Chapin N (1995) An entropy metric for software maintainability. In: Proceedings of the 28th Hawaii international conference on system sciences, pp 522–523

Chikofsky EJ, Cross JH II (1990) Reverse engineering and design recovery: a taxonomy. IEEE Softw 7(1):13–17

Di Penta M, Germán DM (2009) Who are source code contributors and how do they change? In: 16th working conference on reverse engineering, WCRE 2009, 13–16 October 2009, Lille, France. IEEE Computer Society, Washington, DC, pp 11–20

Di Penta M, Germán DM, Guéhéneuc Y-G, Antoniol G (2010) An exploratory study of the evolution of software licensing. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, ICSE 2010, Cape Town, South Africa, 1–8 May 2010. ACM, New York, pp 145–154

Eick SG, Graves TL, Karr AF, Marron JS, Mockus A (2001) Does code decay? Assessing the evidence from change management data. IEEE Trans Softw Eng 27(1):1–12

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code. Addison-Wesley, Reading

Gall H, Jazayeri M, Krajewski J (2003) CVS release history data for detecting logical couplings. In: IWPSE '03: Proceedings of the 6th international workshop on principles of software evolution. IEEE Computer Society, Washington, DC, pp 13–23

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object oriented software. Addison-Wesley, Reading

Grissom RJ, Kim JJ (2005) Effect sizes for research: a broad practical approach, 2nd edn. Lawrence Earlbaum Associates, Hillsdale

Harrison W (1992) An entropy-based measure of software complexity. IEEE Trans Softw Eng 18(11):1025–1029

Hassan AE (2009) Predicting faults using the complexity of code changes. In: 31st international conference on software engineering, ICSE 2009, 16–24 May 2009, Vancouver, Canada, pp 78–88

Hassan AE, Holt RC (2003) The chaos of software development. In: IWPSE '03: Proceedings of the 6th international workshop on principles of software evolution. IEEE Computer Society, Washington, DC, p 84

Holm S (1979) A simple sequentially rejective Bonferroni test procedure. Scand J Statist 6:65–70

Kuhn A, Ducasse S, Gírba T (2007) Semantic clustering: identifying topics in source code. Inf Softw Technol 49:230–243

Lehman MM (1980) Programs life cycles and laws of software evolution. Proc IEEE 68(9):1060–1076

Lehman MM, Belady LA (1985) Software evolution—processes of software change. Academic, London

Linstead E, Baldi P (2009) Mining the coherence of gnome bug reports with statistical topic models. In: Proceedings of the 2009 6th IEEE international working conference on mining software repositories, MSR '09. IEEE Computer Society, Washington, DC, pp 99–102

Nagappan N, Ball T (2007) Using software dependencies and churn metrics to predict field failures: an empirical case study. In: Proceedings of the first international symposium on empirical software engineering and measurement, ESEM 2007, 20–21 September 2007, Madrid, Spain. IEEE Computer Society, Washington, DC, pp 364–373

Parnas DL (1994) Software aging. In: Proceedings of the international conference on software engineering, pp 279–287

Ratzinger J, Sigmund T, Gall H (2008) On the relation of refactorings and software defect prediction. In: Proceedings of the 2008 international working conference on mining software repositories, MSR 2008, Leipzig, Germany, 10–11 May 2008. ACM, New York, pp 35–38

Shannon CE (1948) A mathematical theory of communication. Bell Syst Tech J 27:379–423, 625–656

Sheskin DJ (2007) Handbook of parametric and nonparametric statistical procedures, 4th edn. Chapman & Hall, London

Thomas SW, Adams B, Hassan AE, Blostein D (2010) Validating the use of topic models for software evolution. In: IEEE international workshop on source code analysis and manipulation. IEEE Computer Society, Los Alamitos, pp 55–64

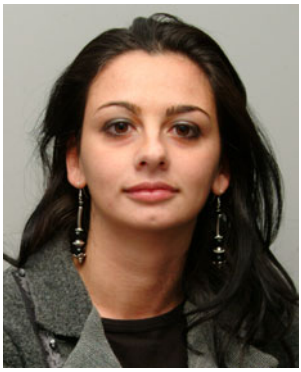
Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST (2006) Design pattern detection using similarity scoring. IEEE Trans Softw Eng 32(11):896–909

van Rijsbergen CJ, Robertson SE, Porter MF (1980) New models in probabilistic information retrieval. In: British Library research and development report, no. 5587. British Library, London

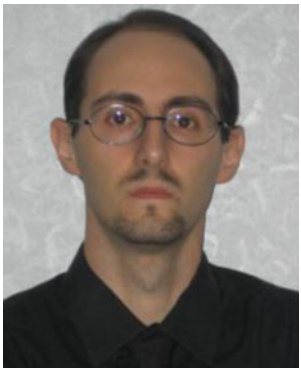
Zimmermann T, Weisgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: ICSE '04: Proceedings of the 26th international conference on software engineering. IEEE Computer Society, Washington, DC, pp 563–572

Gerardo Canfora is a professor of computer science at the Faculty of Engineering of the University of Sannio, Italy. He serves on the program and organizing committees of a number of international conferences. He was general chair of WCRE'06 and CSMR'03, and program co-chair of WETSoM'12 and '10, ICSM'01 and '07, IWPSE'05, CSMR'04 and IWPC'97. Canfora is co-editor of the "Journal of Software: Evolution and Processes" (former: "Journal of Software Maintenance, Research and Practice"); from 2000 to 2004 he was an associate editor for IEEE Transactions on Software Engineering. Canfora authored more than 150 research papers; his research interests include software maintenance and evolution, empirical software engineering, and service-oriented computing. He received the "Laurea" degree in Electronic Engineering from the University of Naples "Federico II".

Luigi Cerulo is assistant professor at the University of Sannio in Benevento (Italy), since 2009, and adjunct researcher at the Institute of Genetic Research "Gaetano Salvatore" (BioGem), since 2009. He received the Laurea degree in Computer Engineering from the University of Sannio in 2001 and the PhD in Software Engineering from the same University in 2006. His research interests include reverse engineering and empirical software engineering with a particular focus on mining software repositories. Since 2007 he is interested in bioinformatics in the area of supervised learning of regulatory relationships between genes. He is the author of almost 40 articles appeared in international journals, conferences, and workshops. He served, and is serving, as program committee member of various software engineering workshops/conferences: IWPSE 2007, WCRE 2008, SCAM 2010, and SCAM 2012. He is member of the IEEE Computer Society and of the Association for Computer Machinery (ACM).



Marta Cimitile is an Assistant Professor of Computer Science at the Faculty of Jurisprudence of the Unitelma-Sapienza University in Rome (Italy). She received a PhD in Computer Science in 2008 at the Department of Informatics at the University of Bari. She received a "laurea" degree with full marks and honors in Ingegneria Gestionale in 2003 from the University Federico II of Napoli, presenting a thesis in "Enterprise management: organizational and technical problems related to implementation of a CRM". She has collaborated with the University of Bari from April-October 2004. Her main research is in the study and evolution of knowledge management, knowledge transfer and data mining. In the last year, she was involved in several industrial and research projects and was author of several publications about the above topics.



Massimiliano Di Penta is associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, reverse engineering, empirical software engineering, search-based software engineering, and service-centric software engineering. He is author of over 150 papers appeared in international conferences and journals. He serves and has served in the organizing and program committees of over 60 conferences such as ICSE, FSE, ASE, ICSM, ICPC, CSMR, GECCO, MSR, SCAM, WCRE, and others. He is program co-chair of ICSM 2012 and has been general chair of SCAM 2010, WSE 2008, general co-chair of SSBSE 2010, WCRE 2008, and program co-chair of SSBSE 2009, WCRE 2006 and 2007, IWPSE 2007, WSE 2007, SCAM 2006, STEP 2005, and of other workshops. He is steering committee member of ICSM, CSMR, WCRE, IWPSE, and past steering committee member of ICPC and SCAM. He is in the editorial board of the Empirical Software Engineering Journal edited by Springer, and of the Journal of Software: Evolution and Processes edited by Wiley. He is member-at-large of the executive committee of the Technical Council of Software Engineering (TCSE). He is member of IEEE, IEEE Computer Society, and of the ACM.