An Exploratory Study of Factors Influencing Change Entropy

Gerardo Canfora*, Luigi Cerulo**, Massimiliano Di Penta*, Francesco Pacilio***
$*$ Dept. of Engineering-RCOST, University of Sannio, Italy
$**$ Dept. of Biological and Environmental Studies, University of Sannio, Italy
$***$ Sytel Reply Srl, Italy
canfora@unisannio.it,lcerulo@unisannio.it,dipenta@unisannio.it,f.pacilio@reply.it

***Abstract*—Software systems continuously change for various reasons, such as adding new features, performing bug fixing, or doing some refactoring activities. Such changes may either increase the source code complexity and disorganization, or help to reduce it. Developers apply adequate design principles and assets, including design patterns, to make software resilient to changes and control complexity.**

This paper empirically investigates the relationship of source code complexity and disorganization—measured using source code entropy—with three factors: different kinds of changes occurring to software systems, the presence of design patterns in the source code, and the number of contributors that modified the source code file.

Results of an exploratory study carried out on an interval of the life-time span of two open source systems, ArgoUML and Eclipse-JDT, suggest that (i) different kinds of changes—namely refactorings and other kinds of changes—may contribute either negatively or positively to the entropy, (ii) the use of design patterns does not necessarily help to mitigate code degradation—thus confirming previous findings on the role played by design patterns—and (iii) entropy tends to increase with the number of file committers.

Keywords: Software evolution, software entropy, design patterns, empirical study.

I. Introduction

It is well-known that software systems are continuously subject to maintenance activities with the purpose of introducing new features to comply with user needs or market pressure, fix faults, or adapt the system to new environments and architectures. Due to the time pressure, the limited effort, and the lack of a disciplined process, these activities tend to deteriorate the software system structure, increasing source code complexity, negatively affecting the system design, and, in general, making the system more difficult to be understood and maintained in the future. Parnas [1] 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 [2], [3], state that:

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

Reengineering, redocumentation and refactoring are a few examples of activities aimed at mitigating, or even reversing, the effects of aging. The application of sound design principles, including the use of design patterns, aims at preventing aging by delaying the code decay.

Design patterns, i.e., recurring design solutions for object-oriented systems [4], are an example of a design practice aimed at slowing down the aging process. On the one hand, design patterns are expected to introduce several advantages, such as increased reusability, and improved maintainability and comprehensibility. On the other hand, recent studies also investigated risks concerned with the use of design patterns, including the change-proneness of design patterns [5], [6] and their roles [7] involved in crucial parts of a software system, the fault-proneness of some design patterns [8], and the correlation between design patterns fault-proneness and their generated crosscutting concern scattering degree [9].

This paper investigates the relationships between the complexity of a software system over time and (i) different kinds of changes occurring to the system, ranging from changes aimed at performing refactoring/restructuring/clean-up activities, and other changes aimed at fixing faults or introducing new features; (ii) the presence of design patterns in the software system; and (iii) the number of developers who committed changes to a given file up to when the complexity was observed. To quantify the level of complexity and disorganization of the source code, this paper stems from previous work [10], [11], [12], [13] that adapted Shannon’s entropy [14] to measure the entropy of an evolving Software system.

The study analyzed the history of two open source software systems, ArgoUML and Eclipse-JDT. Results show that, as expected, changes aimed at fixing bugs or introducing new features tend to increase the entropy, while refactoring decreases it. Also, we found that classes participating in design patterns do not exhibit a lower entropy than other classes, and that the entropy increases with the number of developers changing a file.

The paper is organized as follows. Section II reviews the definition of software entropy provided in literature, Section III describes the data extraction process used for this study and Section IV defines the empirical study. Results are reported and discussed in Section V. Section VI discusses the threats to validity. After a discussion of the related literature (Section VII), Section VIII concludes the paper and outlines directions for future work.

II. Software Entropy

In this section we recall the definition of software entropy introduced by Hassan [13]. The definition stems from the 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* [14]. 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, $p(x_k) = 1$, the entropy is minimal.

The basic code change model proposed by Hassan [13] considers a software system as a *data emitter*, where source file modifications are assumed to be 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 software entropy within a period of time P, e.g. a week, where a file f_i underwent a certain number of changes $chg_P(f_i)$, is defined as:

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

where the probability mass function of f_i, i.e. probability that a source file changes in such a period of time, is estimated as the ratio between $chg_P(f_i)$ and the total number of changes in that period for all files. Instead of source files, other units of code can be used, such as functions or code chunks. The choice of files is based on the belief that a source file is a conceptual unit of development where developers tend to group high coupled entities.

Figure 1 shows an example of how the change entropy of a file is computed. The system is composed of 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.

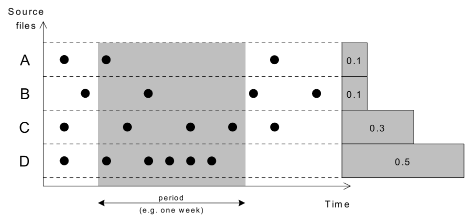


Figure 1. Complexity of a Change Period [13]. It is computed (right-side bars) as the number of changes to a file in a period divided by the total number of changes in the period.

Software 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 [13], the definition of software 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.

III. Data Extraction Process

This section describes the five steps followed to extract the data used in our study.

Step 1: Extraction of data from versioning systems: the first step aims at extracting change information from the Concurrent Versions Systems (CVS)/SubVersioN (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.

Step 2: Computing the entropy of changes: the second step aims at computing the change entropy of each file in a given set of subsequent commits. We use the definition of Hassan [13] recalled in Section II. In particular, we compute the probability that a source code file changed in a period in which a total of 500 commits occurred. We used such a number as (i) it was previously used by Hassan [13] and it is not too short nor too long compared to the total number of file commits for the systems we analyzed (36316 for ArgoUML and 71389 for Eclipse-JDT, see Table I in Section IV); (ii) it is large enough to limit changes occurring across two subsequent periods. We preferred to use modification-based periods rather than time-based periods, as the latter would treat similarly periods of relative inactivity and intense development periods. The probability that a file changed in a period of observation is computed as the number of commits to the file in that period, divided by the number of commits that all files underwent in the same period.

Other than computing the entropy, we are interested to determine whether a file, in the period following the time window when it underwent a change, had a relative increase or decrease of its change entropy. Thus, given the entropy

$e_{i,k}$ of file i in the period k, its percentage of relative variation $\Delta e_{i,k}$ is computed as:

$$\Delta e_{i,k} = 100 \cdot \frac{\Delta e_{i,k+1} - \Delta e_{i,k}}{\Delta e_{i,k}}$$

Step 3: Identifying refactoring-related changes: as mentioned in the introduction, we are interested to investigate whether changes related to refactoring influence the entropy differently than other kinds of changes. It is important to note that, in this context, we are not strictly interested to object-oriented refactoring [15], 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.* [16]—aimed at identifying refactorings by inspecting CVS/SVN commit notes and mining strings likely describing refactoring activities, e.g., "refactoring" or "cleanup" or "removed unneeded". This approach does not guarantee to identify all refactorings, e.g., it misses refactorings not mentioned in the commit notes. At least, this approach would allow for identifying a set of commits that, according to what declared by the developers, are related to refactorings. To this aim, one author of our paper scrutinized the commit notes for the systems we analyzed, classifying them into notes related to refactorings and other notes, and another author double-checked the classification, to minimize the presence of false positives.

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 related the 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 [17]. 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 entropy in the CVS/SVN repository. Nevertheless, future work will aim at investigating whether 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.* [18], following exactly the same detection process (and data) used in our previous work [6], [9].

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 [4]—but also modified versions (variants). The analysis has been performed using the tool[1] 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, 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[2].

Once having identified 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.

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.

IV. 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 presence of design patterns in source code, and the number of committers that changed a file—relates with the code entropy. The *quality focus* is the source code maintainability and comprehensibility, that could be affected by the increasing code entropy. The *perspective* is of researchers interested to investigate to what extent the change entropy can be controlled by means of refactoring activities, by using proper design assets, or by controlling the number of people modifying a file. The *context* consists of two open source systems, ArgoUML, and Eclipse-JDT, having a different size: the first is a medium-sized system, the second is a large one.

[1]http://java.uom.gr/∼nikos/pattern-detection.html
[2]https://javacc.dev.java.net/

Table I
CASE STUDY HISTORY CHARACTERISTICS.

SYSTEM	COMMITTERS	FILE COMMITS	REFACT COMMITS (%)	RELEASES	KNLOC	CLASSES
ArgoUML	42	36316	9039 (25%)	0.9–0.20	99.5–159.5	801–2373
Eclipse-JDT	41	71389	3617 (5%)	1.0–3.0	205.5–534.4	2089–6949

Table II
DETECTED DESIGN PATTERNS
(FM: FACTORY METHOD, P: PROTOTYPE, S: SINGLETON, AC: ADAPTER-COMMAND, C: COMPOSITE, D: DECORATOR, O: OBSERVER, SS: STATE-STRATEGY, TM: TEMPLATE METHOD, V: VISITOR)

SYSTEM	DESIGN PATTERNS										TOTAL
	CREATIONAL			STRUCTURAL			BEHAVIORAL				
	FM	P	S	AC	C	D	O	SS	TM	V	
ArgoUML	0–7	0–9	76–174	7–27	1	6–15	5–9	5–48	12–33	0	117–256
Eclipse-JDT	47–54	0–6	21–45	106–270	1–4	16–25	11–33	168–242	63–109	0–71	564–831

ArgoUML[3] is an open source UML modeling tool with advanced software design features, such as reverse engineering and code generation. The project started in September 2000 and is still active. We considered an interval of observation ranging from September, 2000 (release 0.9.0) to December 2005 (release 0.20 ALPHA 4), where 58 releases have been produced including alpha, beta, and release candidates. The SVN repository contained, for that period, 36316 file changes (25% of which classified by us as refactorings) performed by 42 committers. The total number of detected design patterns grew from 117 to 255, for some of them—Factory-Methods, Singletons, and Adapter-Commands—such number oscillates over the time. No Visitor design pattern was detected in ArgoUML.

Eclipse-JDT is a set of plug-ins that adds the capabilities of a full-featured Java IDE to the Eclipse[4] platform. *Eclipse-JDT*, as any Eclipse project is organized into releases, stable, integration, and nightly builds. We analyzed the time interval between November 2001 and June 2004, when Eclipse 3.0 was released. In such a period, the CVS repository contained 71389 file changes (5% of which classified by us as refactorings) performed by 41 committers. The number of detected design patterns reached the maximum of 831 at release 2.1.

Table I reports the main characteristics of the two systems, including (i) the number of project committers; (ii) the number of file commits and how many of them were related to refactoring activities, (iii) the range of releases analyzed, and (iv) the system size range in terms of non-commented lines of code (KNLOC) and classes. Table II reports the ranges of design patterns detected in various releases of the two systems.

A. Research Questions

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

- **RQ1:** *How do refactoring changes affect the entropy if compared with other kinds of changes?*
- **RQ2:** *How does the source code entropy vary between classes participating and not to design patterns?*
- **RQ3:** *How does the entropy relate to the number of contributors to a class?*
- **RQ4:** *How do the interaction among the factors investigated in RQ1, RQ2, RQ3 influence a class entropy??*

B. Variable Selection

This section describes the dependent and independent variables this study aims at investigating on. They are computed as described in Section III. 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 participation (or not) of a class contained in a source code file to *design patterns*, and, if this is the case, to what design patterns it participates;
- *the number of committers* that modified a file up to a given period when the entropy is estimated.

The dependent variables are:

- *the entropy* of changes occurring to a source code file. We mainly relate this variable to the participation of a class contained in a file to design patterns and to the number of committers that modified a file. In these cases, we are interested to observe whether the two factors influence the file change entropy, rather than to observe whether the entropy changes in correspondence of a particular event.
- *the entropy variation* for a file between one period of 500 commits and the following one. We relate this dependent variable mainly with kinds of changes, as we are interested to understand whether a certain kind of change increases or reduces the entropy in the near future.

C. Analysis Method

To address **RQ1**, we compare the entropy variation for refactorings and other changes using the (non-parametric) Mann-Whitney test. In addition, we use the Cohen d [19] effect size measure to understand whether such a difference

[3]http://argouml.tigris.org
[4]http://www.eclipse.org

is of practical interest. The Cohen d effect size indicates the magnitude of a main factor treatment effect on the dependent variables. For independent samples, it is defined as the difference between the means (M_1 and M_2), divided by the pooled standard deviation ($\sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2}$) of both groups: $d = (M_1 - M_2)/\sigma$. The effect size is considered small for $0.2 \le d < 0.5$, medium for $0.5 \le d < 0.8$ and large for $d \ge 0.8$. Finally, we graphically show the 95% intervals of confidence for entropy variations.

For **RQ2**, we again use the Mann-Whitney test and Cohen d effect size to compare the entropy of classes participating and 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 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 Bonferroni correction, i.e., dividing the threshold p-value (0.05) by the number of possible combinations of pairwise comparisons (i.e., among all design patterns).

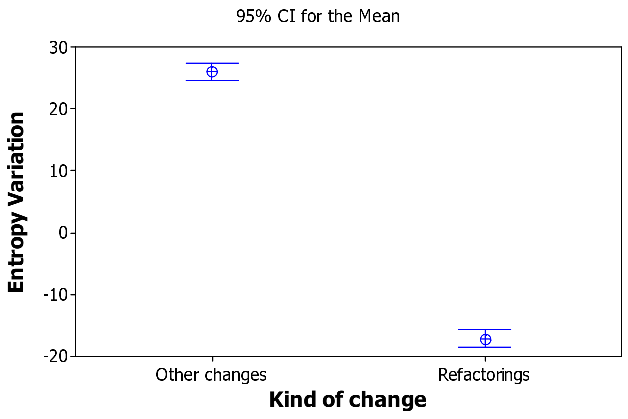
For **RQ3**, we graphically analyze, using boxplots, how the entropy increases for changes occurring to files that had, up to when the change occurred, a different number of contributors. Then, we analyze if there is or not an increasing trend of the entropy when the number of contributors increases, using the Augmented Dickey Fuller (ADF) test (H_0: the series has a trend).

RQ4 aims at analyzing the interaction among different factors affecting the entropy or inducing different entropy variations. We do not analyze all possible combinations of factors, but rather a subset of them we believe are more meaningful. Specifically, we investigate whether: (i) the kind of changes interacts with the participation of a class to design patterns, with respect to the entropy variation induced by the changes; and (ii) the participation of a class to design patterns interacts with the number of committers for the file where the class is defined, with respect to the file change entropy. To analyze such interactions, we use the two-way analysis of variance (ANOVA) as well as interaction plot, which plot lines connecting average values of the dependent variable for different combinations of the independent variables. Line intersections indicate the presence of interactions.

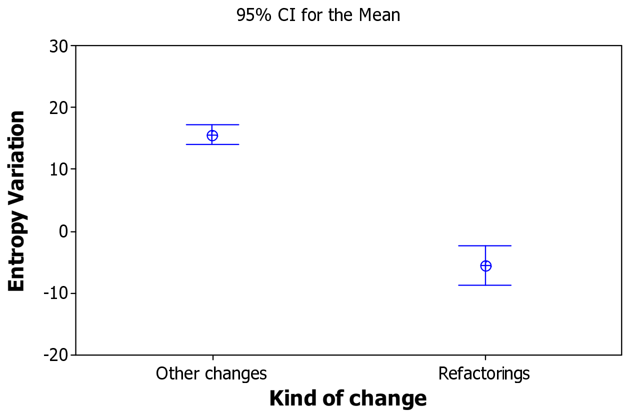
V. Results

This section reports results of our empirical study to answer the research questions formulated in Section IV-A. Data for verification/replication are available on-line[5].

[5]http://www.rcost.unisannio.it/mdipenta/entropy-rawdata.tgz



(a) ArgoUML



(b) Eclipse-JDT

Figure 2. Entropy variation for different kinds of changes: interval plots (95% confidence intervals).

Table III
ENTROPY VARIATION: DESCRIPTIVE STATISTICS AND MANN-WHITNEY TEST FOR REFACTORING AND OTHER CHANGES.

System	Stat.	Refact. changes	Other changes
ArgoUML	Mean	-17.12	25.97
	Std. dev.	66.26	122.39
	M-W p-value		< 0.01
	Eff. size		0.38
Eclipse-JDT	Mean	-5.58	15.57
	Std. dev.	96.98	209.23
	M-W p-value		0.33
	Eff. size		0.10

A. RQ1: How do refactoring changes affect the entropy if compared with other kinds of changes?

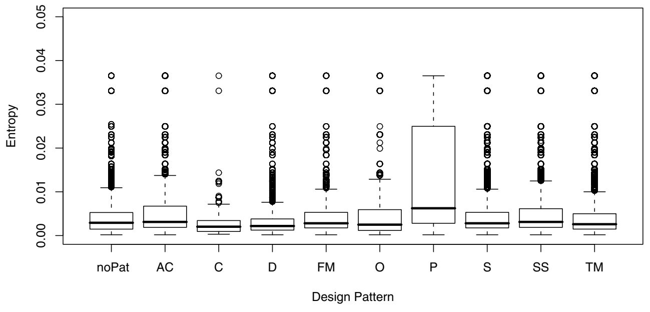
Figure 2 reports confidence intervals of entropy variations (Δe) in periods following changes related to refactorings and to other changes. Descriptive statistics are reported in Table III, together with results of Mann-Whitney test and Cohen d effect size. For both systems, it is possible to observe that, on average, the entropy exhibits a negative

Table IV
EXAMPLES OF COMMIT NOTES FOR REFACTORING ACTIVITIES.

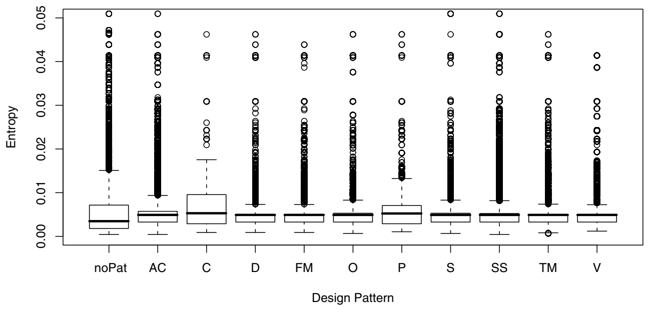
ARGOUML
- Style issues. Privatised stuff. - Refactoring of the Model component Issue 2696. This part is making the Factory and Helper interfaces accessed from the Model class. . . . - Replaced deprecated log4j Category with Logger. - Some cleanups in the use of the Model interfaces. - Created a getter 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

Table V
ENTROPY DESCRIPTIVE STATISTICS AND MANN-WHITNEY TEST FOR CLASSES PARTICIPATING AND NOT IN DESIGN PATTERNS.

System	Stat.	Pattern Yes	Pattern No
ArgoUML	Mean	0.00544	0.00419
	Median	0.00312	0.00293
	Std. dev.	0.00751	0.00475
	M-W p-value		< 0.01
	Eff. size		0.19988
Eclipse-JDT	Mean	0.00653	0.00640
	Median	0.00491	0.00347
	Std. dev.	0.00607	0.01806
	M-W p-value		< 0.01
	Eff. size		0.00992



(a) ArgoUML



(b) Eclipse-JDT

Figure 3. Entropy for changes in classes participating and not in design patterns.

variation after refactorings, and a positive variation after other changes, and this is more evident in ArgoUML. For ArgoUML, the Mann-Whitney test indicates a significant difference of Δe between refactorings and other changes, with a small effect size (d=0.38), while the difference is not significant in Eclipse-JDT, and the effect size negligible.

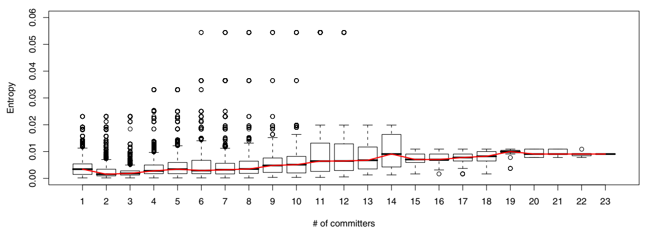
Table IV shows some examples of commit notes related to refactoring activities, taken among the most frequent ones. ArgoUML refactoring notes are very informative, and are related to (i) making resources private, thus increasing the information hiding, (ii) re-organizing package interfaces (this, again, would minimize the impact of changes), (iii) more generally, re-conceiving part of the architecture, replacing old structures with new, better designed ones. Eclipse-JDT refactoring notes provide less information, but, again, appear to be related to interface cleanup, increase of information hiding, and code re-organization.

B. RQ2: How does the source code entropy vary between classes participating and not to design patterns?

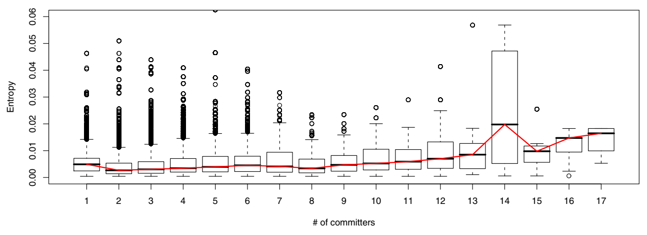
Table V reports descriptive statistics of the entropy for classes participating in design pattern and for other classes, and compares them using the Mann-Whitney test and the Cohen d effect size. The table shows that the entropy is slightly higher for classes participating in design patterns. Although there is a significant difference, the effect size is small for ArgoUML and negligible for Eclipse-JDT. This, overall, suggests that the presence of design patterns do not play a tangible role in reducing or keeping the change entropy low.

Figure 3 shows boxplots of the entropy for different design patterns (the *noPat* boxplot is for classes not participating in design patterns). For ArgoUML, the Kruskal-Wallis test indicates a significant difference of change entropy among classes participating in different design patterns (p-value <0.01). Specifically, there is a significantly higher entropy for classes participating in the Prototype (P) design pattern (p-value <0.01), with a large effect size (d=0.68). Instead, Composite (C) and Decorator (D) exhibit a significantly lower entropy (p-value <0.01), with a low effect size (d=0.20). The Prototype design pattern is mainly used to "make a system independent of how its products are created" [4]. Specifically, in ArgoUML they are mainly used to create figures representing UML diagrams. Every time there is need to update the diagrams, Prototypes classes are impacted.

For Eclipse-JDT, again the Kruskal-Wallis test indicates a significant difference of change entropy among classes participating in different design patterns (p-value <0.01). In this case, the Composite (C) exhibits the highest entropy (p-value <0.01, d=0.43), followed by the Prototype (P). The Composite allows a transparent use of individual objects and



(a) ArgoUML



(b) Eclipse-JDT

Figure 4. Entropy for classes with different number of committers.

composite objects made of various objects of a hierarchy. In Eclipse-JDT, Composites are used to represent the document object model of the Abstract Syntax Tree (AST). Changes occurring to various analyzers might require to modify the AST representation, e.g., to add attributes for storing some specific information.

It is interesting to note that the above results well complement other findings obtained when studying design patterns of ArgoUML and Eclipse-JDT. In fact, in a previous work [9] we found that in ArgoUML the Prototype design pattern has a relatively high scattering degree that significantly correlates (Spearman rank correlation=0.79) with the code fault-proneness. The Prototype is among the most frequently changed patterns in ArgoUML, and above all it mainly underwent changes affecting class interfaces [6], thus likely to have a high impact. In Eclipse-JDT, the Composite has a high scattering degree, and there is a high correlations (Spearman rank correlation=0.89) of the scattering degree with the code fault-proneness.

C. RQ3: How does the entropy relate to the number of contributors to a class?

Figure 4 shows how the entropy varies when the number of committers for a file increases. For both systems, an increase of the entropy is clearly visible, although, while for ArgoUML it decreases between 1 committer and 2 committers, and then it starts to increase, for Eclipse-JDT the entropy starts to increase for a number of committers greater than eight. For ArgoUML, the ADF test reveals the presence of an increasing trend (p-value=0.81), and the entropy increases of 163% from 1 to 23 committers. For Eclipse-JDT, there is also an increasing trend (p-value=0.70), and the entropy increases of 235% from 1 to 17 committers.

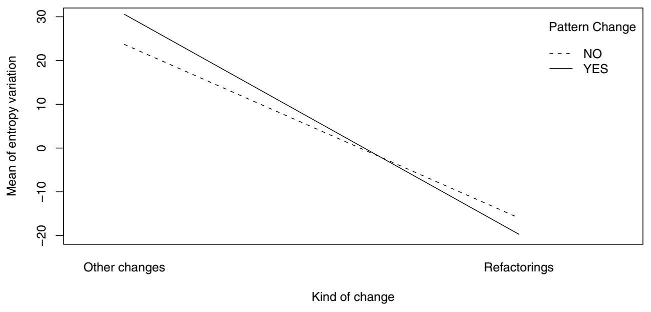
By looking at the files that, overall, had the highest number of committers, we noticed that, for ArgoUML, the class having the highest number of committers is *GeneratorJava* (22 committers). Such a class generates the Java code that appears in property tabs of each UML model class/object. Changes to different UML models—performed by different authors—will likely impact this class. The classes *ProjectBrowser* and *Project* have 21 and 20 committers respectively; the former manages the user interface for project browsing, the latter represents the model class where project information is kept. Changes to various project-related information will affect these classes. Other classes with a high number of committers are: *TabProps* (19 committers), which manages the visual tabs showing the properties of an object, and *Actions* (18 committers), which handles the various user interface actions.

For Eclipse-JDT, the class *JavaEditor* has the highest number of committers (17), followed by *JavaPlugin* (16), then by *IJavaHelpContextIds* and *CompilationUnitEditor* (both 15), and finally by *JavaOutlinePage* (14) which exhibits a change entropy spike. *JavaEditor* is the main class of the Java text editor, therefore it is likely that many changes (performed by different committers) to editing features of Eclipse-JDT can impact this class and increase its entropy. *JavaPlugin* governs the access to the various features (workbench, menus, editors, etc.) of JDT, thus again it might be impacted by changes related to different features. Similar considerations apply to *IJavaHelpContextIds* (which handles contextual help) and *CompilationUnitEditor*, which again is related to the Java editor. Last, but not least *JavaOutlinePage* is the Java editor outline, that is impacted by most of the changes occurring to the Java editor, thus explaining the entropy spike.

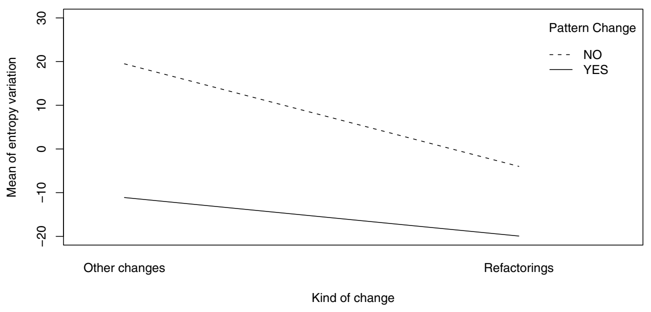
D. RQ4: How do the interaction among the factors investigated in RQ1, RQ2, RQ3 influence a class entropy?

Finally, we analyze whether there is an interaction among the different factors (related to the change entropy) we analyzed: kind of change, whether or not a class participates in design patterns, and the number of committers that modified the file up to the change for which the entropy was computed.

Figure 5 shows interaction plots of the Δe by kind of change and by participation in design patterns. For ArgoUML, the figure shows that refactoring changes contribute to decrease the entropy more than other changes; however this effect is more evident on design pattern classes than for other classes. A two-way ANOVA confirms what Figure 5-a shows, i.e., a significant effect on the entropy variation of the kind of change, of the participation (or not) of the class in design patterns, and a significant interaction of the three factors (p-value <0.01 in all three cases).

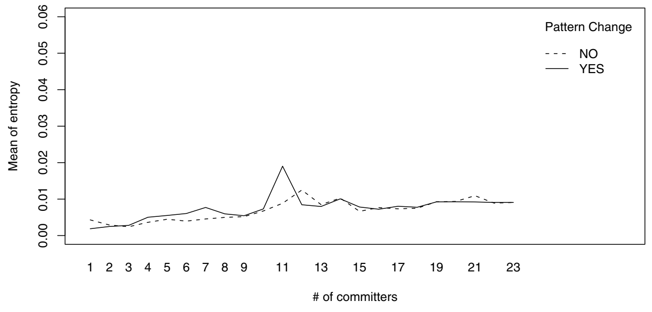


(a) ArgoUML

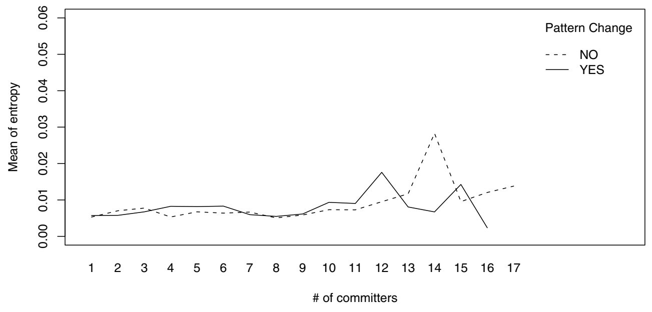


(b) Eclipse-JDT

Figure 5. Interaction plot of entropy variation by kind of change and participation of classes in design patterns.



(a) ArgoUML



(b) Eclipse-JDT

Figure 6. Interaction plot of entropy by number of committers and participation of classes in design patterns.

For Eclipse-JDT, changes occurring to classes participating in design patterns always exhibit a lower entropy variation than other classes, although for other classes refactorings help to decrease the entropy more than for classes participating in design patterns. Very likely, this happens because, due to the well-modularized structure of Eclipse, making a high use of hierarchies and adopting a plugin-based architecture, design pattern classes have a low level of entropy already, that cannot be reduced any further. ANOVA confirms such results, indicating a significant effect of both factors—kind of change and participation in design patterns—(p-value < 0.01), however it indicates no significant interaction among them (p-value=0.20), as can be noticed from Figure 5-b also (lines do not cross each other).

Figure 6 shows interaction plots for what concerns the effect on the entropy of the participation of classes in design patterns, of the number of committers, and of the interactions of these two factors. For ArgoUML, the two-way ANOVA by number of committers and participation in design patterns indicates a significant influence of the two factors on the entropy, and a significant interaction between them (p-value < 0.01 in all three cases). In fact, as Figure 6-a shows, the (plain) line for classes participating in design patterns tends to be above the (dashed) line for other classes (i.e., regardless of the number of committers, the entropy of design pattern classes is higher, with some exceptions).

For Eclipse-JDT, the two-way ANOVA by number of committers and participation in design patterns indicates a significant influence of the number of committers on the entropy (p-value < 0.01), no significant effect, as discussed in **RQ2** of the participation of classes in design patterns (p-value=0.21), although a significant interaction of the two factors (p-value=0.01). Overall, the two lines for classes participating in design patterns and other classes tend to be overlapped. For some values of the number of committers the entropy tend to be higher for classes not participating in design patterns, while the other way around happens in other cases, however this mainly depends on particular entropy values of some specific classes (as discussed in **RQ2**, for instance there are only a few classes with a very high number of committers).

VI. 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. They are mainly due to imprecision in the measurements we performed. While we carefully checked the identified refactorings to avoid false positives, we could not make any claim about the recall of the refectoring identification approach used. Thus, our results will be representative of the identified sample only. However, in that case our aim was rather to maximize the precision. Design pattern detection is another source of imprecision and of limited recall. Tsantalis *et al.* [18] report a precision of 90% or above for the systems they analyzed. Committers might represent, as discussed in [17] a partial view of who modified a file. Future work will aim at considering all file contributors other than simply committers. Finally, to compute the entropy, we used a modification-based period consisting of 500 changes as it was done in previous studies; however, we have not experimented yet different period sizes, nor different ways of determining the period (e.g., time-based approaches).

Conclusion validity concerns the relationship between the treatment and the outcome. Attention was paid to not violate assumptions made by statistical tests. We mainly used non-parametric tests, plus ANOVA, which is pretty robust to deviation from normality. Other than testing the presence of significant differences, we also checked the presence of practically relevant differences using the Cohen d effect size.

Threats to *internal validity* concern factors that can influence our observations. It must be clear that the study does not claim any cause-effect relationship between the investigated independent variables (kind of changes, presence of design patterns, and number of committers) and the dependent variable (entropy and its variation). At least, we discussed our results and looked at the system characteristics and source code trying to provide interpretations to our findings, also relating them with findings of previous studies performed on the same systems. Finally, we are aware that, although we believe that the factors we investigated are relevant factors influencing the entropy, these are not the only ones, and there may be others we have not taken into consideration.

Threats to *external validity* concern the generalization of our findings. We analyzed two systems of different size and having different characteristics, that indeed were reflected in our results. However, it is highly desirable to replicate this initial study on further systems.

VII. Related Work

This section describes related work for what concerns (i) the use of entropy as a measure of disorder in software and (ii) the effect of design patterns on software change-proneness and fault-proneness.

The proposal of entropy-based metrics to measure the disorder of a software system is not recent. Chapin [11] proposed the use of such a metric to suggest maintenance activities when the entropy begins to grow. Harrison [12] proposed the entropy as a measure of software complexity. Bianchi *et al.* [10] 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.

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 [20] 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 the system is, the more complex the systems code becomes over time. In a subsequent work, Hassan [13] 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 [13] for code changes. However, we show that some kinds of changes, like refactorings (in a broader sense) contribute to decrease the entropy, and investigate the relationship between the entropy and other factors, such as the presence of design patterns and the number of committers.

In recent years, several papers analyzed the effect of design patterns on software change-proneness and fault-proneness. Bieman *et al.* [5] analyzed four small size systems and one large size system to identify the observable effects of the use of design patterns, such as pattern change-proneness. Vokáč [8] analyzed the corrective maintenance of a large commercial product over three years, comparing defect rates for classes that participated in design patterns versus those that did not participate.

Previous studies performed on the relationship between software changes and design patterns [6] or design pattern roles [7] suggested that there is a relation between different kinds of design patterns and the frequency of changes occurring on software systems; however, this also depends, as we found in this paper, on specific characteristics of the software systems analyzed, and to what extent design patterns play a role in crucial features of the system. A recent study [9] indicates that design patterns induce crosscutting concerns that, in turn, have a significant correlation with the software fault-proneness. Noticeably, this is particularly true for design patterns for which we noticed a high entropy, specifically Prototypes in ArgoUML and Composite in Eclipse-JDT. As shown in Section V, many insights we got when studying the relation between change entropy and design patterns confirmed (and complemented) previous findings about design pattern change-proneness, fault-proneness, and the kinds of change design pattern classes underwent.

VIII. Conclusion and Work-in-Progress

Software evolution activities—as claimed in many literature studies [2], [3], [1]—tend to increase the software aging and complexity which, stemming from information theory [14], some authors measured in terms of entropy [10], [11], [12], [13].

This paper reported an exploratory study aimed at investigating how various factors affect positively or negatively the source code change entropy. Specifically, we considered factors related to the system structure, and in particular the presence of design patterns as identified by a reverse engineering tool [18], as well as factors specific of the software project activities, i.e., the number of committers that modified a file, and whether or not a change was related to refactorings. Results showed that refactorings contribute to significantly decrease the code change entropy, while other changes tend to increase it. Design patterns, while often help to make the code more resilient to changes, do not have a major positive effect on the entropy, as they often trigger many, scattered changes [9]. As expected, files having a higher number of committers tend to have a higher entropy.

Future work aims at extending this study on further systems, at better understanding the causes of entropy variations, and in general at considering other factors that might have an effect on the source code change entropy.

References

[1] D. L. Parnas, “Software aging,” in *Proceedings of the International Conference on Software Engineering*, 1994, pp. 279–287.

[2] M. M. Lehman, “Programs life cycles and laws of software evolution,” *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, September 1980.

[3] M. M. Lehman and L. A. Belady, *Software Evolution - Processes of Software Change*. Academic Press London, 1985.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[5] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander, “Design patterns and change proneness: An examination of five evolving systems,” in *9th International Software Metrics Symposium (METRICS03)*. IEEE Computer Society, 2003, pp. 40–49.

[6] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, “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*. New York, NY, USA: ACM Press, 2007, pp. 385–394.

[7] M. Di Penta, L. Cerulo, Y.-G. Gueheneuc, and G. Antoniol, “An empirical study of the relationships between design pattern roles and class change proneness,” in *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*. IEEE, 2008, pp. 217–226.

[8] M. Vokáč, “Defect frequency and design patterns: An empirical study of industrial code,” *IEEE Trans. Software Eng.*, vol. 30, no. 12, pp. 904–917, 2004.

[9] L. Aversano, L. Cerulo, and M. Di Penta, “The relationship between design patterns defects and crosscutting concern scattering degree: an empirical study,” *IET Software*, vol. 3, no. 5, pp. 395–409, October 2009.

[10] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio, “Evaluating software degradation through entropy,” in *METRICS ’01: Proceedings of the 7th International Symposium on Software Metrics*. Washington, DC, USA: IEEE Computer Society, 2001, p. 210.

[11] N. Chapin, “An entropy metric for software maintainability,” in *Proceedings of the 28th Hawaii International Conference on System Sciences*, 1995, pp. 522–523.

[12] W. Harrison, “An entropy-based measure of software complexity,” *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 1025–1029, 1992.

[13] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada*, 2009, pp. 78–88.

[14] C. E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, pp. 379–423, 625–56, 1948.

[15] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Publishing Company, 1999.

[16] J. Ratzinger, T. Sigmund, and H. Gall, “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, May 10-11, 2008*. ACM, 2008, pp. 35–38.

[17] M. Di Penta and D. M. Germán, “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*.

[18] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, “Design pattern detection using similarity scoring,” *IEEE Trans. Software Eng.*, vol. 32, no. 11, pp. 896–909, 2006.

[19] J. Cohen, *Statistical power analysis for the behavioral sciences (2nd ed.)*. Hillsdale, NJ: Lawrence Earlbaum Associates, 1988.

[20] A. E. Hassan and R. C. Holt, “The chaos of software development,” in *IWPSE ’03: Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2003, p. 84.