An Empirical Investigation of Changes in Some Software Properties Over Time

Joseph Gil
The Technion—Israel Institute of Technology
yogi@cs.technion.ac.il

Maayan Goldstein, Dany Moshkovich
IBM Research—Haifa
{*maayang,mdany*}*@il.ibm.com*

***Abstract*—Software metrics are easy to define, but not so easy to justify. It is hard to prove that a metric is valid, i.e., that measured numerical values imply anything on the vaguely defined, yet crucial software properties such as complexity and maintainability.**

This paper employs statistical analysis and tests to check some plausible assumptions on the behavior of software and metrics measured for this software in retrospective on its versions evolution history. Among those are the reliability assumption implicit in the application of any code metric, and the assumption that the magnitude of change, i.e., increase or decrease of its size, in a software artifact is correlated with changes to its version number.

Putting a suite of 36 metrics to the trial, we confirm most of the assumptions on a large repository of software artifacts. Surprisingly, we show that a substantial portion of the reliability of some metrics can be observed even in random changes to architecture. Another surprising result is that Boolean-valued metrics tend to flip their values more often in minor software version increments than in major increments.

I. Introduction

Many hidden assumptions are made in the study and management of software evolution: for example, there is an almost universal agreement on a Dewey like version numbering scheme, and people tend to believe that

> *changes to major version number are correlated with the magnitude of the software change* **(major-changes-large)**

It is likewise common knowledge that

> *software changes fall in a continuous range between the evolutionary end, at which most meaningful properties are preserved, and the antipodal revolutionary end* **(evolutionary-revolutionary-spectrum)**

(The concepts of revolutionary and evolutionary changes do not have crisp definition, yet, the distinction should be clear; restructuring an existing software system to fit a model–view–controller pattern [1] would be considered by many as revolutionary change, and of plugging in a new encryption module to a banking system as an evolutionary change.)

Further, it is plausible to assume that

> *most releases of new software versions are evolutionary* **(mostly-evolutionary)**

and that

> *revolutionary changes tend to coincide with changes to major version number* **(revolutionary-changes-in-major-versions)**

We may also subscribe to beliefs regarding the kind of changes. One would tend to think that

> *additions to an existing software body tend to follow existing style; more so with evolutionary changes* **(preservation-of-style)**

Also, it is believable that

> *even large changes to software tend to leave substantial isolated portions of the code unchanged* **(locality-of-change)**

One of the questions that motivated this work regarded the correctness of a assumption that anyone who uses code metrics is inclined to make (recall the large number of software metrics (see e.g., [2]–[8]):

> *metrics are reliable* **(metrics-reliability)**

The statement that in a certain software system 22% of the classes are `final` while 8% are `abstract`, is meaningless, unless there is a certain degree of persistence (in the course of software evolution) to the marking of a class as final or abstract.

Are these assumptions indeed true? Can they be verified? This paper reports on a large scale empirical study for checking such assumptions. Our results confirmed most of the assumptions. But, a number of very "believable" assumptions, including **locality-of-change** were refuted.

Our study employed 36 different code metrics, selected from several independent collections of metrics. We organized these metrics in a taxonomy whose main groups are: *marker* (or Boolean) metrics, *local numerical* metrics and *global numerical* topological metrics. To understand better what we mean by "topological" metrics, recall that programs can be readily represented as a directed graph of classes, packages, or other modules, which can then be subjected to graph theoretical algorithms [9]–[11].

To check the different assumptions, we used a large corpus of software versions, and applied the same set of metrics to each version. We then asked whether the results are reliable, i.e., whether the values obtained in a certain version are

978-1-4673-1761-0/12/$31.00 © 2012 IEEE

MSR 2012, Zurich, Switzerland

predictive of the values in the subsequent version. We further investigated how changes in version size and number are correlated with the metrics. Finally, we examined how some assumptions change for different groups of metrics.

An intriguing finding of this work is that a substantial portion of the reliability of the global metrics can be observed even if random perturbations are applied to the architecture. This means, in a sense, that these metrics do not capture an inherent architectural property of the software.

Another interesting result is that marker metrics tend to change less in major version increments and more in minor version changes. This may mean that major version releases are more stable and carefully organized than the minor ones.

Contributions. The main contributions of this paper are:

1) Raising the somewhat less visited issue of *software metrics reliability*.
2) An introduction of a taxonomy of code metrics.
3) The discovery of similarity in many of the properties of metrics in each group.
4) A systematic application of statistical methods to confirm (or refute) assumptions on software.
5) The revelation that local metrics are highly reliable, i.e., their values are preserved as the software evolves.
6) The discovery of the surprising fact that local metrics tend to change more often in minor version changes.
7) The discovery of the link between the ranking imposed by numerical global metrics and the topological architecture, i.e., software properties which can be inferred by examining the structure of the software graph, but without using any semantical information.
8) The discovery that although global metrics tend to be reliable, much of this reliability is due to the limited scope of changes.

The remainder of this article is organized as follows. The data corpus and the way it was selected are described in Section II. Section III then analyzes the size changes of software artifacts present in the corpus, examining assumptions **locality-of-change**, **evolutionary-revolutionary-spectrum**, **mostly-evolutionary** and **major-changes-large**. This section takes an intermission to remind the reader of Kendall's tau correlation coefficient and its use as an indicator of similarity between rankings.

Section IV presents our metrics suite and the way it was selected, and our metrics taxonomy. In Section V we study the reliability of the metrics that yield Boolean values, discussing the **mostly-evolutionary**, **evolutionary-revolutionary-spectrum** and **preservation-of-style** assumptions. These assumptions are revisited in Section VI which discusses the numerical metrics. The analysis that shows that at least part of the reliability of global numerical metrics cannot be attributed to inherent "software architecture" is presented in Section VII. Section VIII concludes and suggests directions for further research.

II. Software Corpus

A. Artifacts

The software corpus used in our experiments comprised 19 software *artifacts*, drawn from the *Qualitas Corpus*[1], a colossal collection of Java software used extensively in many empirical software engineering studies.

These artifacts included: the Java compiler, `javac`, `ant` (Java's equivalent of `make`), and `junit` (the Java unit testing library), Eclipse's JDT core, search, and SWT, `Antlr` (a framework for constructing compilers, interpreters, etc.) `hibernate`, `log4j`, `struts`, and others.

B. Versions

For each artifact, we analyzed a number of *versions* from the corpus. We harvested all available versions of each of the artifacts, omitting only three versions in which global renaming made it difficult to automatically trace classes of previous versions. In total, our corpus comprised 95 versions.

The essential size characteristics of the corpus are summarized in Table I. The corpus totaled some 78 thousands types (classes and interfaces), organized in 5,500 packages. The number of types in the versions selected in the corpus spans two orders of magnitude (42 through 6,444), with a median and average of a few hundreds of types. A similar variety is observed in the number of packages.

Table I
Size characteristics of the software corpus.

Size Metric	Mean	Median	Min	Max	Total
Types	$822 \pm 1,125$	420 ± 285	42	6,444	78,099
Packages	58 ± 98	23 ± 13	3	469	5,500
Edges	$3,767 \pm 4,910$	$2,069 \pm 1,437$	77	27,764	357,897

Each software version was modeled as a *directed graph*, in which types serve as nodes, and edges lead from a type to all types which it uses directly, i.e., inheriting from it, declaring a variable of it, invoking one of its methods, etc. Edges leading to outside the artifact, e.g., the edge that leads from almost every Java class to `java.lang.Object`, were ignored. The number of edges thus found is shown in the last row of the table.

Table I introduces a $\pm$ notation that embellishes the mean with the standard deviation, e.g., the *mean* number of types is 822 (averaged over all 95 software versions), while the standard deviation is $1,125$. Similarly, the median is embellished with the *median absolute deviation* (M.A.D.), defined as the median of the absolute deviations from the median of the distribution.

The large standard deviation and the wide range of values are not surprising—software varies greatly in size. For this

[1]See the *Qualitas Research Group, Qualitas Corpus* `http://www.cs.auckland.ac.nz/~ewan/corpus`

reason, we prefer the median and the M.A.D. as a pair of summarizing statistics over the mean and standard deviation. Admittedly, the median and the M.A.D are less efficient statistical measures than the mean and the standard deviation, but they are robust to outliers, which are unavoidable with this great variety.

C. Pairs

Our study of software change was carried by organizing the 95 versions in the corpus in an ensemble of 76 *pairs* of subsequent versions of the same artifact.

Some statistics of software growth and the extent of preservation in the pairs of the corpus are shown in Table II.

Table II
GROWTH AND CHANGES IN CONSECUTIVE ARTIFACT VERSIONS.

Metric	Mean (%)	Median (%)	Min (%)	Max (%)
Types	137±64	112±11	100	517
Edges	146±92	118±17	90	712
Remaining Types	91±16	97±3	19	100
Continuing Types	75±23	79±17	11	100
Remaining Edges	86±18	94±6	23	100
Continuing Edges	71±26	76±20	6	100
Unchanged Types (outgoing)	17±7	17±5	1	36
Unchanged Types (incoming)	16±7	17±4	1	36
Unchanged Types (both)	16±7	17±5	1	36

On average, the number of types increased by 37% and the number of edges by 46%. Again, we observe a wide spectrum of changes, e.g., in one of the pairs, the number of edges increased by a factor of 7. There were even cases in which the number of edges decreased, probably thanks to code refactoring which reduced coupling between types.

This great variety can be interpreted as a supportive indication of **evolutionary-revolutionary-spectrum**. Furthermore, the fact that in the first two lines of Table II, the median is smaller than the mean, is consistent with the assumptions that most changes are evolutionary, and that evolutionary changes typically incur smaller size changes. However, this raw data does not provide sufficient grounds for the correct placement of any given pair between evolutionary and revolutionary extremes.

The next group of rows in Table II shows the statistics of the ratio of types (resp. edges) that are common to both versions of a pair, compared to the total number of types in earlier version (*remaining*) and the later version (*continuing*). We have that the mean fraction of remaining types is 91%, while the median fraction is 97%. The fact, recurring across the entire group, that the median is greater than the mean, is, again, consistent with the assumption that most of the pairs represent a more evolutionary change, in which most of the types remain in the subsequent version. The fewer pairs which represent revolutionary changes bring the mean lower than the median.

Note that it could be the case that some of the types which were marked as removed by our analysis tools, were simply renamed. The extent of this analysis error is bounded above by the (small) number of removed types.

Finally, the last rows of the table summarize the percentage of the classes for which none of the incoming (outgoing) were changed during the evolution process. We learn that about 17% of the relations to- or from- types stay unchanged.

We can say that the *functionality* of one in six types does not change, at least in the sense that the set of other types it uses does not change. Also, one in six types does not change its *duty* in two subsequent versions of an artifact, at least as far this duty is judged based on the set of other types it serves. Conversely, **locality-of-change** is not confirmed by these results, changes to software typically border with 5 out of 6 types.

To summarize, the typical topological change between two subsequent versions of a software system is characterized by:

1) a preservation of almost all types and edges;
2) a preservation of the locale of about one sixth of the types;
3) an increase of about 10% in the number of types and of about 20% in the number of edges.

III. SIZE CHANGES OF ARTIFACTS IN THE CORPUS

A. Correlating Magnitude Changes with Version Number Changes

Our study of the correlation between magnitude changes and version number changes, begins with the introduction of a notion of *version number change cardinality*, which is assigned to each pair of artifact versions by comparing the version numbers (as assigned by the artifact numbering scheme) of the pair members: a cardinality of $1/2^{n-1}$ is associated with a change to a n^{th} level version number. Thus, the cardinality of a change to the major version number is 1; a change to the second level version number, (e.g., versions `1.3` and `1.4`) has cardinality $1/2$, etc.

Our ensemble comprised 12 pairs of change cardinality 1, 36 pairs of cardinality $1/2$, 19 pairs of cardinality $1/4$, 8 pairs of cardinality $1/8$, and 1 pair of cardinality $1/16$.

Figure 1 now depicts the distribution of relative changes in the number of edges and types in the corpus' pairs. Each circle in the figure corresponds to a pair in the corpus; larger circles corresponding to more cardinal changes.

We do not know whether the more modest changes are evolutionary, that is whether these changes tend to preserve existing properties. However, the picture depicted in Figure 1 is at least consistent with **evolutionary-revolutionary-spectrum** and **mostly-evolutionary**: in most pairs, the increase in the number of types (edges) is modest; notwithstanding, a non-vanishing number of the pairs exhibit substantial increases to the number of types (edges).

Are the more drastic changes linked to the publication of major new editions of the software? It is difficult to confirm or refute **major-changes-large** by visual inspection of

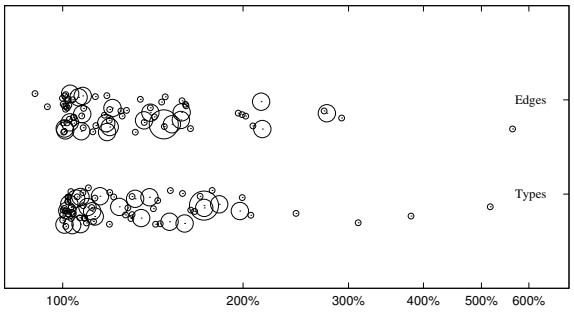


Figure 1. Distribution of relative changes in the number of edges and the number of types.

Figure 1, in trying to determine whether the larger circles tend to show on the left or on the right of the figure.

Instead, we shall describe an analytical method for studying this correlation using *Kendall's tau coefficient* [12]. The coefficient gives a measure of the agreement between two different rankers of the same data set. Given is a set of elements, and their relative ranking by two rankers. The coefficient can be used for measuring the agreement of the ranking of a certain metric in two versions of the software. It assumes its maximal value of 1 in the case of full agreement, and its minimal value of -1 is achieved in the case of total disagreement. The set for comparison is that of the types which occur in both versions.

Thus, Kendall's coefficient is similar to Pearson correlation, except that it is non-parametric, rendering it applicable to our "change cardinality" ranking (which is ordered, but has no obvious, non-arbitrary mapping to numerical values) with magnitude of change. Herein, we used a version of the coefficient denoted τ_b which deals with ties (e.g., two version pairs with the same cardinality of change). The underlying statistical test assigns a statistical confidence level to each value of τ_b.

Statistical Test of ***major-changes-large*** In comparing the ranking of the pairs by the (relative) magnitude of change and the change cardinality we found that $\tau_b = 0.35$ (resp. $\tau_b = 0.30$) when the size of the increase is measured in types (edges) with p-value < 0.001, thus confirming **major-changes-large**.

Our values of τ_b were computed across all pairs, ignoring the concern that increment to the second level version number in one artifact may be as drastic as a major version number increment in another. In restricting the comparison to versions of the same artifact, we found even higher values that are statistically significant, e.g., $\tau_b > 0.60$. (Notwithstanding, artifacts with small number of versions did not yield statistically significant values.)

B. Characteristic of Change

We now present a topological breakdown of changes to the software graph. This breakdown will be used below (Section VII) to guide the generation of a random mutation of a given software version, and for examining the reliability of metrics against these mutations.

Fix a pair of two consecutive versions of an artifact. Then, three kinds of types can be distinguished: *core* types are those that are present in both versions, *removed* types are those that are present in the early version but not in the later version, and, conversely, *new* types are those that are present in the newer version only. Also, changes to edges can be further characterized: removed edges are either *core/core*, *core/removed*, *removed/core* or *removed/removed*. Added edges are either *core/core*, *core/new*, *new/core*, or *new/new*. Preserved edges are always of the *core/core* kind.

Table III summarizes the statistics of the changes according to this breakdown. All values in the table are obtained by first normalization of the absolute numbers and then computing the median. Normalization of the absolute number of edges (or types) was with respect to the number of edges (or types) in the early version.

Table III
BREAKDOWN OF ADDED AND REMOVED EDGES IN THE CORPUS (MEDIAN VALUES, NORMALIZED)

Edges Kind	Edges (%)	From Types (%)	To Types (%)
Core/Core (preserved)	94±6	95±4	64±12
Core/Core (added)	4±2	9±6	8±4
Core/Core (removed)	3±2	8±5	6±3
Core/New	3±3	7±6	7±5
New/Core	9±8	13±12	13±8
New/New	7±7	19±17	14±13
Core/Removed	0±0	0±0	0±0
Removed/Core	1±1	2±2	3±3
Removed/Removed	0±0	1±1	1±1

For example, the first row of the table shows that 94% of the edges between core types of an early were preserved when progressing to a newer version. Those edges originated (had them as sources) in 95% of the types; as their targets the edges used 64% of the types in an early version of software artifact.

The mid section of Table III reveals an interesting (but not too surprising) property of the "graph cut" separating the old and the new portions of software: the largest bulk of added edges are those that connect newly introduced types to core types. Edges in the opposite direction—leading from core types to newly introduced types—are rare.

IV. METRICS AND THEIR TAXONOMY

There are hundreds if not thousands metrics described in the literature. In our experiments, we tried to cover a variety of metrics and give ample consideration to the most popular ones. This section describes the 36 software metrics we used, and proposes a taxonomy of metrics of this sort.

A. Criteria for Classification of Software Metrics

Given is G, the directed graph of software system, where each node v represents a module of this system, and an

edge $e(s,t)$ leads from a *source* node s to a *target* node t if type s uses type t. A *metric* then is a function μ_G (or just μ if G is clear from the context) that assigns a value $\mu(v)$ to each node $v \in G$.

Metric nature. If $\mu(v)$ depends solely on the topology of G, we say that μ is *topological*. In contrast to topological metrics stand *semantical* metrics whose value takes into account a deeper analysis of the node contents (by e.g., examining the code in this node), and the sort of the edges incident on it (by e.g., distinguishing between different kinds of dependencies among nodes). The suite includes 17 semantical metrics.

Metric scope. Another criterion for classification is whether $\mu(v)$ depends on G in its entirety, rather than on a restricted neighborhood of v. We say that a metric is *strictly local* if $\mu(v)$ does not change with changes to G that preserve incoming and outgoing edges to v (along with the identity of the nodes at the other end of these). In other words, metric μ is strictly local if $\mu(v)$ depends solely on v and its neighbors. Also, μ is *local*, if for every $v \in G$ there is a set of nodes $S \subsetneq G$, such that $\mu(v)$ does not change despite arbitrary changes to G, as long as the nodes $S \cup \{v\}$ and the edges among these are intact.

For example, the widely studied Chidamber and Kemerer (CK) suite [2] has a number of strictly local methods, including *Number of Children* (NOC) and *Coupling between Object Classes* (CBO). The *Depth in Inheritance Tree* (DIT) metric however is local, but not strictly local.

Obviously, local metrics are more suited to the study of a single type, or a small portion of the code; this kind of metrics is not expected to be telling much of the architecture.

Overall, we have 14 local metrics. A sub-category of *local* metrics (10 metrics in our suite) is that of internal metrics; a metric μ is *internal* if $\mu(v)$ depends only on v. A local metric does not make sense unless it is semantical. *Weighted Methods Per Class* (WMC) [2], is an example of an internal metric. A metric which is not local is *global*.

Metric range. Our third criterion for classifying metrics is based on the type of values they yield; *continuous* metrics (e.g, PageRank) yield real values, while *discrete* metrics (e.g., CBO, NOC, and DIT) yield integers, typically drawn from a small range, say $o(|G|)$. We have 3 continuous metrics, and 19 discrete metrics. The remaining metrics belong to a special kind of discrete metrics henceforth called *markers*, which yield Boolean-, that is true- or false-, values.

B. Metrics Used in the Experiments

Table IV enumerates the metrics used in our experiments, classifying these according to this taxonomy.

Marker Metrics. The first fourteen metrics in the table are markers: *final*, *abstract* and *interface* are simply the JAVA class attributes with the same name.

Next comes a group of four topological metrics. The *sink* marker is assigned to types from which a bottom-up study of a software system may start since they are referred by any other type in the system (either directly or indirectly). Conversely, the *source* marker is for types from which a top-down study may start. The *balloon* marker [13] is for types which have only one client, i.e., nodes whose in-degree is 1. And, the *wrapper* marker is just the opposite—nodes whose out-degree is 1. Next, we have a group of micro-patterns markers [14].

Chidamber and Kemerer Metrics. The next six metrics are all semantical, undirected, discrete, and local; they were all drawn from Chidamber and Kemerer's suite [2].

Plain Topological Metrics. The next local metric is *#Incoming*, which counts the number of immediate clients a type has. In contrast, *#Clients* is a global metric defined as the total number of clients of a type, including both immediate and non-immediate clients. *#Outgoing* and *#Descendants* are the dual of these two. Observe that *#Descendants* is identical to Page-Jones and Constantine's [15, Chap. 9] *encumbrance* metric, which, according to the first author of this book, is indicative of the "sophistication" of a type, its role and may even be predictive of its fate.

Strongly Connected Components Metrics. The next group of metrics is computed from the directed acyclic graph of *strongly connected component*s of G. Recall that there is a directed path between any two nodes that reside in the same

Table IV
METRICS USED IN EXPERIMENTS AND THEIR CATEGORIES.

Metric	Nature	Scope	Range
final	semantical	internal	Boolean
abstract	semantical	internal	Boolean
interface	semantical	internal	Boolean
sink	topological	local	Boolean
source	topological	local	Boolean
baloon	topological	local	Boolean
wrapper	topological	local	Boolean
pure	semantical	internal	Boolean
pool	semantical	internal	Boolean
designator	semantical	internal	Boolean
function pointer	semantical	internal	Boolean
stateless	semantical	internal	Boolean
sampler	semantical	internal	Boolean
canopy	semantical	internal	Boolean
DIT	semantical	local	discrete
NOA	semantical	local	discrete
NOC	semantical	local	discrete
CBO	semantical	local	discrete
RFC	semantical	local	discrete
WMC	semantical	local	discrete
#Incoming	topological	local	discrete
#Clients	topological	global	discrete
#Outgoing	topological	local	discrete
#Descendants	topological	global	discrete
#SCCIncoming	topological	global	discrete
#SCCClients	topological	global	discrete
#SCCOutgoing	topological	global	discrete
#SCCDescendants	topological	global	discrete
SCCSize	topological	global	discrete
#DominatedBy	topological	global	discrete
#DominatorHeight	topological	global	discrete
#DominatorWeight	topological	global	discrete
PageRank	topological	global	continuous
Betweeness	topological	global	continuous
Belonging	semantical	local	continuous

strongly connected component; this theoretical structure of a graph makes sense in a software context since all types in such a component are interdependent, and hence should probably be studied together.

In our suite, *SCCSize* represents the size of the strongly connected component (SCC) that a type belongs to. *#SCCIncoming* and *#SCCClients* are, respectively, the number of SCC's immediate and indirect clients. Their duals are *#SCCOutgoing* and *#SCCDescendants*.

Dominators Tree Metrics. The penultimate metrics group is computed from the *dominators tree* of G. Recall that a node r dominates a node v, if the only way of getting from into v is through r, and that there is an edge in this tree if r is the "most immediate" dominator of v. Thus, the dominators tree is likely to identify pivotal points of the software system. From this tree we compute the *#DominatedBy* metric which is the number of nodes that dominate this node, the *#DominatorHeight*, which is the height of the node in the dominators tree, and *#DominatorWeight*, giving the number of nodes that a given node dominates.

Other Metrics. In the last group of metrics in Table IV we have Google's *PageRank* and *Betweenness*, yet another measure of graph centrality [16]; roughly speaking, nodes that occur on many shortest paths connecting other nodes have higher Betweenness value than those that do not.

The last metric in the table is *Belonging* used, e.g., in JDepend[2], which estimates the extent by which a type belongs to its package by dividing the number of edges it has (both incoming and outgoing) to other types in the package by the total number of edges incident on the type.

V. Marker Metrics

Table V gives the essential statistics of the prevalence of the marker metrics in the suite.

Table V
ESSENTIAL STATISTICS OF THE PREVALENCE OF THE MARKER METRICS IN THE SUITE.

Metric	Mean (%)	Median (%)	Min (%)	Max (%)
final	15.0 ± 15.1	7.3 ± 6.9	0.0	48.5
abstract	4.6 ± 2.4	4.1 ± 1.8	0.8	11.8
interface	10.1 ± 5.2	8.1 ± 4.2	1.7	21.2
sink	1.3 ± 2.2	0.6 ± 0.6	0.0	16.7
source	27.7 ± 16.4	29.5 ± 13.7	1.0	55.3
balloon	10.8 ± 8.9	7.2 ± 4.6	1.1	42.1
wrapper	25.7 ± 7.1	23.9 ± 3.0	12.0	49.5
pure	8.9 ± 5.0	8.1 ± 3.7	0.8	21.2
pool	1.4 ± 1.2	1.0 ± 0.6	0.0	5.9
designator	0.4 ± 0.6	0.2 ± 0.2	0.0	4.3
function pointer	0.2 ± 0.4	0.0 ± 0.0	0.0	2.2
stateless	28.7 ± 8.8	29.3 ± 4.9	9.8	53.0
sampler	0.9 ± 0.7	0.8 ± 0.3	0.0	3.2
canopy	17.1 ± 9.1	15.9 ± 7.8	3.4	47.6

The first group of markers in the table are JAVA type attributes. As we see, about 4% of all types are *abstract*, about 10% are interfaces, and 15% are *final*. The variance of the *final* attribute is greater than that of *abstract*, with some versions not using it at all, while others using it for almost half of the classes. Later we will see that this simple architecture discovery hints are surprisingly reliable.

The next group in Table V is of topological markers. The prevalence of sinks is low, but still could be explanatory of the architecture of the underlying software. About one in three types is a source in our corpus. This is explained by the large number of frameworks and libraries in our data set. The resilience of these two markers to the teeth of time tends to be high, as we shall see shortly.

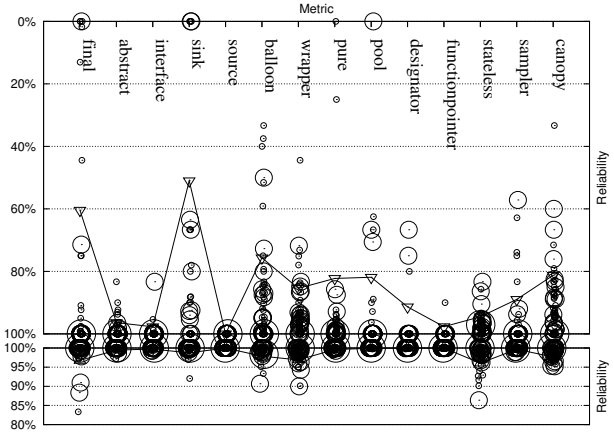
Types with only one client (balloons) are quite popular (10%), but much more popular are wrapper types, i.e., those types make use of only one type in the artifact.

Micro patterns are the third group in the table. Of these, it is surprising to see that almost one third of the types in the corpus are *stateless*. Even though 10% of the types in the corpus are interfaces, about 20% of real classes are stateless, i.e., do not manage state variables at all.

Overall, we see that there is a substantial variety in prevalence of each of metrics. The prevalence of pure types, for example, ranges between 0.8% and 21.2%. Even the smallest standard deviation, 0.4% for the function pointer micro pattern, is large compared to its 0.2% average prevalence.

A. Reliability

The top part of Figure 2 depicts the reliability values of marker metrics in the corpus: columns correspond to the metrics, while each circle on a column corresponds to a certain pair of consecutive versions. The circle height represents the reliability of the marker metric in this pair, that is, the portion of types that preserve this marker as the software evolves from the earlier to the later version.



Larger circles denote more cardinal version changes; triangles denote the mean minus standard deviation barrier.

Figure 2. Reliability of marker metrics (top) and their negation (bottom) in the corpus.

[2] http://clarkware.com/software/JDepend.html

We see that reliability is generally high; In considering, for example, the `final` marker, we observe that ***(i)*** in the vast majority of pairs, fewer than 5% of all `final` classes lose this property as the software evolves, but still, ***(ii)*** there are pairs in which the loss of the `final` property occurs in 30%, 50% and even 100% of the cases.

More generally, we have that ***(i)*** in the majority of pairs, marker metrics are extremely reliable (the median reliability is always 95% or higher, and is greater than 99% for all but two metrics); however, ***(ii)*** in a non-negligible number of pairs, a large portion of the types lose their marking.

The bottom part of Figure 2 depicts similar information as the top, except that it pertains to the negation of marker metrics, e.g., the first column of circles in the figure represents the relative number of classes that were *not* `final` in the earlier version, but became `final` in the later version. Even though negations are more reliable, phenomena ***(i)*** and ***(ii)*** can still be discerned.

The numerical results support these observations: the median reliability is always 99% or higher; it is around 100% for the vast majority of metrics. This, together with the fact that the mean reliability is 92% or higher, and it is most often greater than 98% confirm **metrics-reliable**.

Also, together, ***(i)*** and ***(ii)*** confirm **mostly-evolutionary**. Assumption **evolutionary-revolutionary-spectrum** is confirmed by visual examination of the spectrum of reliability values in each of the 28 columns present on Figure 2: there is no obvious dichotomic discretization of these spectra.

To confirm **revolutionary-changes-in-major-versions**, we computed τ_b of the reliability values and the pair's change cardinality for each of the marker metrics (and their negations). Similarly to the testing in Section III, this computation was carried out for the entire ensemble, but also separately for the set of versions of each of the artifacts in the ensemble.

Rather surprisingly, the results revealed a *positive* correlation of reliability values and cardinality. In other words, we found that marker metrics tend to change less in major version increments and more in minor version changes.

The specifics are as follows: half of the τ_b values were statistically significant (p-value less than 0.05); in all of these τ_b was positive, ranging between 0.13 and 0.33. Further, even all non-significant values were positive (with the sole exception of the `interface` marker metric in which $\tau_b = -0.03$). The same happens for τ_b values computed within each artifact. All statistically significant values are positive, ranging from $\tau_b = 0.49$ and $\tau_b = 1$. (Again, artifacts with a small number of versions did not usually have statistically significant results).

B. Style Preservation

Having studied changes to *existing* types in our corpus, we now turn to checking whether newly added classes tend to follow the "style" of existing software body, and whether this tendency is correlated with the cardinality of the change. The difficulty in testing **preservation-of-style** is that the arsenal of standard statistical tests is good at showing that a set of values *does not* follow a given (null-hypothesis) distribution, but usually falls short of showing the inverse—that the values' set indeed *obeys* a given distribution.

We applied the standard χ^2 test for each of the marker metrics and each of the pairs in the corpus (total of 76×14 tests) to determine whether there is any statistical difference in the prevalence of the metric in the earlier software version and its prevalence among newly added types.

Statistically significant values of the χ^2 value were found in only 25.5% of the tests, and, in confirmation of **preservation-of-style**, these significant changes to the prevalence are correlated with the cardinality of the change. This correlation is not so strong, $\tau = 0.09$, but it is statistically significant (p-value < 0.01).

VI. Numerical Metrics

Each of the numerical metrics defines a relative ordering of the classes present in a software version graph. We therefore use Kendall's τ_b coefficient (see Section III) as the measure of numerical metrics reliability. The computation of τ_b is (naturally) done only for the types which are present in both versions. However, the value of a metric μ_{G_i} may depend on types which only occur in G_i.

High values of τ_b imply high reliability. For example, if $\tau_b = 0.9$ for a certain numerical metric μ and a certain pair of versions $\langle G, G^* \rangle$, then the implication

$$\mu_G(u) > \mu_G(v) \Rightarrow \mu_{G^*}(u) > \mu_{G^*}(v)$$

holds for 95% of types $u, v \in G \cap G^*$.

We thus computed the values of τ_b for all metrics. This computation was restricted to consecutive pairs only for two reasons: first, the reliability value in moving from version i to version $i+2$ can be broken down to, at least mentally, to two factors: that of the progression from version i to version $i+1$ and that of the progression from version $i+1$ to version $i+2$. Second, the consideration of all pairs biases our sample towards artifacts with more versions.

Table VI presents the essential statistics of these values for *local* numerical metrics.

Table VI
ESSENTIAL STATISTICS OF τ_b OF LOCAL METRICS ACROSS CONSECUTIVE VERSIONS OF SOFTWARE ARTIFACTS.

Metric	Mean	Median	Min	Max
DIT	0.95 ± 0.08	0.98 ± 0.02	0.55	1.00
NOA	0.96 ± 0.06	0.98 ± 0.02	0.70	1.00
NOC	0.94 ± 0.09	0.97 ± 0.03	0.34	1.00
CBO	0.93 ± 0.07	0.95 ± 0.04	0.59	1.00
RFC	0.93 ± 0.07	0.94 ± 0.04	0.58	1.00
WMC	0.93 ± 0.07	0.94 ± 0.04	0.56	1.00
#Incoming	0.94 ± 0.08	0.96 ± 0.04	0.52	1.00
#Outgoing	0.93 ± 0.06	0.94 ± 0.04	0.67	1.00
Belonging	0.87 ± 0.13	0.88 ± 0.07	0.22	1.00

Most obviously, all values are high. In fact we have that the average value of τ_b, computed across all metrics and all pairs, is 0.93, with high confidence level (p-value < 0.001), in confirmation of **metrics-reliable**. We also see that the median is greater than the mean, confirming **evolutionary-revolutionary-spectrum** and **mostly-evolutionary**.

Similar finding are exhibited by Table VII which repeats Table VI for *global* metrics. Interestingly, the mean and median values of all metrics are quite similar. Still, in comparing Mean, Median, and Min columns in the two tables we see that global metrics are (slightly) less reliable than local metrics.

Table VII
ESSENTIAL STATISTICS OF τ_b OF GLOBAL METRICS ACROSS CONSECUTIVE VERSIONS OF SOFTWARE ARTIFACTS.

Metric	Mean	Median	Min	Max
#Clients	0.93 ± 0.09	0.96 ± 0.04	0.55	1.00
#Descendants	0.90 ± 0.10	0.93 ± 0.07	0.63	1.00
#SCCIncoming	0.90 ± 0.12	0.93 ± 0.06	0.48	1.00
#SCCClients	0.92 ± 0.10	0.94 ± 0.05	0.52	1.00
#SCCOutgoing	0.90 ± 0.12	0.93 ± 0.06	0.48	1.00
#SCCDescendants	0.90 ± 0.10	0.93 ± 0.07	0.64	1.00
SCCSize	0.89 ± 0.12	0.93 ± 0.07	0.59	1.00
#DominatedBy	0.87 ± 0.17	0.89 ± 0.09	0.05	1.00
#DominatorHeight	0.87 ± 0.13	0.90 ± 0.08	0.33	1.00
#DominatorWeight	0.87 ± 0.13	0.89 ± 0.08	0.35	1.00
PageRank	0.93 ± 0.08	0.94 ± 0.05	0.50	1.00
Betweeness	0.89 ± 0.11	0.91 ± 0.07	0.41	1.00

To confirm **revolutionary-changes-in-major-version**, we followed the same process as in V. The correlation this time was negative: higher reliability of numerical metrics tends to coincide with more minor increments of the version number (in contrast to the results acquired for marker metrics). Specifically, all τ_b values for the ensemble were significant and negative, and all values for a specific artifact which were significant were also negative. (Values of τ_b ranged between -0.30 and -0.84, but were typically about -0.70.)

VII. UNDERSTANDING RELIABILITY OF GLOBAL NUMERICAL METRICS

How can the high reliability values of global numerical metrics be explained? The answer that we would like to have is: "*these metrics indeed capture the essence of a software architecture; their preservation indicates that architecture is persistent*".

Unfortunately, as it will become clear at the end of this section, this answer is only partially correct. Much of the high agreement of the ranking is explained by the limited scope of changes to software between versions. Even *random mutations* of the software graph reach the same high values.

Simple-minded, random mutation. Let G be a graph of a certain version of a software artifact and let G^* be the graph of the successive version. Then, instead of comparing G with G^* as we did before, we shall compare G now with a mutation graph $M = M(G)$, generated by random mutation of M, which has the same number of types and edges as G^*.

Table VIII shows the advantage of the reliability values found above over random, yet simple minded, graph mutations.

Table VIII
DIFFERENCE BETWEEN RELIABILITY OF GLOBAL METRICS ACROSS CONSECUTIVE VERSIONS OF SOFTWARE ARTIFACTS AND RELIABILITY COMPUTED IN A RANDOM GRAPH GROWTH AND SHRINK.

Metric	Median (Grow)	Median (Shrink)
#Clients	0.35 ± 0.24	0.34 ± 0.21
#Descendants	0.46 ± 0.22	0.46 ± 0.20
#SCCIncoming	0.34 ± 0.26	0.35 ± 0.23
#SCCClients	0.32 ± 0.22	0.37 ± 0.21
#SCCOutgoing	0.34 ± 0.26	0.35 ± 0.23
#SCCDescendants	0.41 ± 0.23	0.40 ± 0.22
SCCSize	0.44 ± 0.26	0.44 ± 0.21
#DominatedBy	0.50 ± 0.27	0.43 ± 0.25
#DominatorHeight	0.45 ± 0.24	0.36 ± 0.23
#DominatorWeight	0.44 ± 0.24	0.35 ± 0.23
PageRank	0.23 ± 0.15	0.21 ± 0.14
Betweeness	0.39 ± 0.14	0.35 ± 0.15

In the experiments, we computed, for each of the metrics, the reliability of the metric in the pair $\langle G, G^* \rangle$ and subtracted from it the reliability of this metric in the pair $\langle G, M_0 \rangle$ where graph M_0 was generated by adding to G random nodes and edges necessary to make it as large as (occasionally as small as) G^*. The results were then summarized in the second table column.

The third table column was computed in a similar fashion, except that it uses a variant method of computing the mutation M_0: instead of growing G, we randomly shrink G^* to obtain M_0, and used the pair $\langle M_0, G \rangle$ instead of $\langle G, M \rangle$.

Comparing both the second and third columns of this table with the high values found in Table VII shows that these high numbers are not so telling. Half to two thirds of the apparent agreement between metric values of two versions of the software is found in totally random mutations.

We observe still that the agreement between metric values in a real successor is *always* better than that of a random mutation. Having this happen *consistently* in 17 metrics cannot be a mere coincidence. We have that with statistical significance of $\alpha < 0.001$ or better, the agreement of metrics ranking is not a matter of pure chance.

Topological mutations. The randomness in the above mutations allowed situations which are unlikely to occur in the life-cycle of software. For example, in selecting edges in a complete random fashion, the number of edges between the existing nodes and the new nodes would be much smaller than in the real new version graph G^*. We ask now whether there exists a more structured mutation that can yield the *same* reliability values as found in actual software evolution.

The five mutations presented next try to imitate the topology of the changes to a software graph. All of these mutations start with the original graph G and apply two preliminary transformations to this graph: First, all edges

and nodes present in G but not in G^* are removed. Second, all nodes present in G^* are added.

These deterministic transformations create a graph which ***(i)*** has all the core nodes, ***(ii)*** has all the preserved core/core edges, and ***(iii)*** has the same number of nodes as G^*. The duty of a subsequent random mutation is to add new edges to this transformed graph so that it has the same number of edges as G^*.

Recall now our categorization of edge kinds and their fate (Section III-B). Edges in graph G^* are of the following five kinds: core/core (preserved), core/core (added), core/new, new/core and new/new. Our construction of the transformed graph is such that the first kind exists in it. All mutations in our experiments add the correct number of missing edges of each of the four remaining kinds.

The difference between the mutations is in the way the source and target nodes are selected for edges in each of these categories. We use three different policies.

1) Same means that the source and the target are not selected at random; we simply copy the edge from G^* to the mutated graph;
2) Random means that the source and the target are selected at random from the sets of all nodes in the corresponding group.
   For example, if we apply this policy of selection to generate random core/new edges, then every such edge connects a random node in the core with a random edge in the new set.
3) Random/boundary is similar to the Random policy, except that the source and the target are selected at random from the more restricted set of nodes which served in G^* as source or target for the corresponding kind.
   For example, in applying this policy of selection to generate random core/new edges, every newly created edges connects *(i)* a node selected at random from the set of core nodes with an edge in G^* leading to a new node with a *(ii)* a node selected at random from the set of new nodes with an incoming edge in G^* starting at a core node.

Table IX uses these policies to describe the mutations we apply. Columns of the table describe the locus of mutations: the ***core*** locus refers to added core/core edges; the ***cut*** locus refers to the core/new and new/core edges; and, the ***new*** locus refers to new/new added edges.

Table IX
MUTATIONS IMITATING A SUBSEQUENT SOFTWARE VERSION

	Core	***Cut***	***New***
M_1	Random	Random	Random
M_2	Random/boundary	Random/boundary	Random/boundary
M_3	Same	Random/boundary	Random/boundary
M_4	Random/boundary	Random/boundary	Same
M_5	Same	Random/boundary	Same

Mutation M_1 is the simplest; it is much like M_0 described above, except that it maintains the balance of edge groups. Mutation M_2 imitates slightly better a real software version, in that it uses the Random/boundary policy for edge selection.

Mutations M_3, M_4 and M_5 were designed with the objective of understanding better which graph locus contributes more to the agreement of metric values:

- Mutation M_5 is almost identical to G^*. The difference is only in a random selection of edges in the ***cut*** graph locus (and, even these edges are not entirely random; they only connect nodes which were adjacent on the ***cut*** in graph G^*).
- Mutations M_3 and M_4 are similar to M_5 except that in M_3 there are random changes to the core locus and in M_4 there are random changes to the new locus.

Results. Figure 3 summarizes the median of the reliability values calculated for the pair $\langle G, G^* \rangle$ and $\langle G, M_i \rangle$ for $i = 0, 1, \ldots, 5$.

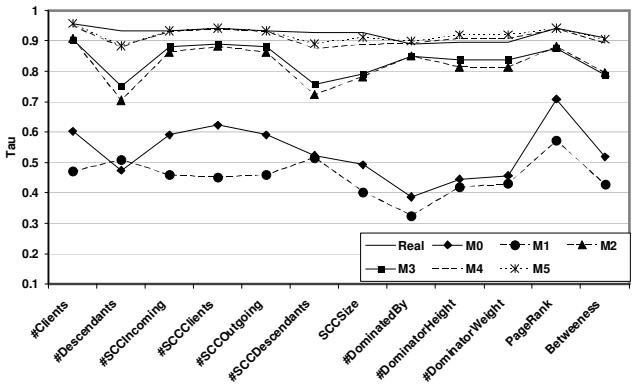


Figure 3. Median value of reliability of global metrics across consecutive versions of software artefact's and random mutations $M_0, \ldots, M_5$.

The top most line in the figure, labeled "real" designates the high agreement values found in the $\langle G, G^* \rangle$ pairs. The other lines correspond to the mutations. We see that the random and not so structured mutations M_0 and M_1 explain about 2/3 of the high values of agreement. Furthermore, even though M_1 is more structured, it is inferior to M_0 in at least several metrics. A substantial improvement occurs when we move to M_2 in which we select an edge to connect two random "portal" nodes.

The reliability values of $\langle G, M_5 \rangle$ are almost exactly the same as those of real software versions, i.e., of $\langle G, G^* \rangle$. This is not very surprising, since the graph M_5 is different from G^*- only in one locus.

Now both M_3 and M_4 are different from G^* in two loci. We expect M_3 to agree with G better than M_4 agrees with it. The reason is that the agreement between metric rankings is compared only at core nodes. As indicated by Table IX, the edges of M_3 at the core are exactly the same as in G^*.

We expected mutation M_3 to yield good reliability values: after all, in this mutation randomness was limited to the ***new*** locus, which is the farthest from the core.

What is surprising though is that mutation M_4 (in which the core/core added edges are random) approximates a real software version almost at the same level of agreement as M_3. Put differently, the ***new*** and the ***core*** loci have the same impact on reliability.

Analysis We have thus observed that the particular way in which real evolution of software "selects" edges in the ***new*** locus has a substantial and measurable impact on the reliability of global numerical metrics. This observation is consistent with **preservation-of-style**.

More importantly, the dual of this observation tells us that our suite of global numerical metrics is sensitive to additions to the ***new*** locus. Only if these additions are done in a manner "consistent" with evolution of real software, reliability is preserved.

VIII. Conclusions

Our study employed a total of 95 versions of 19 software artifacts.

We collected from diverse sources a suite of 36 metrics of JAVA classes. Each of these metrics can be computed directly from the binary representation of the software. Three, almost orthogonal, criteria were proposed for categorizing the metrics: *nature*, *scope*, and *range*.

We introduced a notion of *reliability* for both boolean and numerical valued metrics, and analyzed reliability over 76 pairs of versions of large software artifacts. We found that reliability of all marker (boolean valued) metrics was about 99%.

Reliability of numerical valued metrics was defined using Kendall's tau rank coefficient. We found that reliability of metrics in the same categorical group was similar; local metrics reaching a coefficient of change of about 0.93, and global metrics assuming a coefficient value of about 0.90.

Most of the presumptions presented in Section I were confirmed. Exceptions were: **locality-of-change**, for which we found that 5 out of 6 types included at least one changed type in their neighborhood, and **revolutionary-changes-in-major-versions** whose opposite was confirmed with respect to local metrics.

Assumption **preservation-of-style** was confirmed with our tacit interpretation of the term "style" as the prevalence of marker metrics. In a sense, Section VII tried to explore this assumption from the point of view of global numeric metrics. We showed that the interconnection between newly added types have a strong impact on the ordering of global numeric metrics computed at the core types.

Further research should probably focus on the link between numerical metrics and topological architecture as implied by the phenomena shown in Section VII.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[2] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering.*, vol. 20, no. 6, pp. 476–493, 1994.

[3] M. J. Ordoñez and H. M. Haddad, "The state of metrics in software industry," in *Proc. of the Fifth International Conference on Information Technology: New Generations, (ITNG 2008)*. Las Vegas, Nevada, USA: IEEE Computer Society, Apr. 7-8 2008, pp. 453–458.

[4] G. Lajios, "Software metrics suites for project landscapes," in *Proc. of the 2009 European Conference on Software Maintenance and Reengineering.*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 317–318.

[5] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[6] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[7] N. E. Fenton, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Boston, MA, USA: International Thomson Computer Press, 1996.

[8] B. Neate, W. Irwin, and N. Churcher, "Coderank: A new family of software metrics." in *Proc. of the 17th Australian Software Engineering Conference, (ASWEC 2006)*. Sydney, Australia: IEEE Computer Society, 2006, pp. 369–378.

[9] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[10] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.

[11] R. Sedgewick, *Algorithms*. Addison-Wesley, 1983.

[12] M. G. Kendall and J. D. Gibbons, *Rank Correlation Methods*, 5th ed. Oxford University Press, New York, 1990.

[13] P. S. Almeida, "Balloon types: Controlling sharing of state in data types," in *Proc. of the Eleventh European Conference on Object-Oriented Programming, (ECOOP'97)*. Jyväskylä, Finland: Springer-Verlag, Jun. 9-13 1997, pp. 32–59.

[14] J. Y. Gil and I. Maman, "Micro patterns in java code," *SIGPLAN Not.*, vol. 40, no. 10, pp. 97–116, 2005.

[15] M. Page-Jones and L. L. Constantine, *Fundamentals of object-oriented design in UML*. Boston, MA, USA: Addison-Wesley, 2000.

[16] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.