How Much Information Do Software Metrics Contain?

Yossi Gil [*]
The Technion—Israel Institute of Technology
yogi@cs.technion.ac.il

Maayan Goldstein Dany Moshkovich
IBM Research–Haifa
maayang@il.ibm.com mdany@il.ibm.com

Abstract

Software metrics computation and presentation are considered an important feature of many software design and development tools. The *System Grokking Technology* developed by IBM research enables investigation, validation and evolution of complex software systems at the level of abstraction suitable for human comprehension. As part of our ongoing effort to improve the tool and offer more useful abstractions we considered adorning the presented information with software metrics. The difficulty in doing that is in selecting among the legions of metrics competing for both scarce screen space and for the architect's attention. In this paper, we describe a new criterion for evaluating the competing metrics based on a normalized version of Shannon's information theoretical content. We also give values of these in a large software corpus and for a large set of metrics.

Based on our measurements and this criterion, we can recommend the presentation of two metrics: module centrality, as measured by a variant of Google's classical page ranking algorithm, and module size, as measured by Chidamber and Kemerer's WMC metric.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—Complexity measures, Software science

General Terms Design, Measurement

Keywords Metrics, Information

1. Introduction

The total volume of software that the human race produces (and consumes) increases at a staggering rate. With this increase, and the accompanying blowup in software complexity [24], more and more automatic tools are developed by academia and industry [1, 2, 13, 16, 22, 23, 29] to aid in understanding, validating and evolving of existing software.

The *System Grokking technology* (SGT) [9, 12] is our own such tool. It supports software architects in the incremental and iterative user driven understanding of software systems through higher levels of abstraction. The technology includes a visual modeling framework for the representation of domain-specific software elements and the relationships between these in a graphical manner. SGT was applied to a spectrum of programming languages, including Java, C, C++ and Cobol, and in a variety of application domains. This was made possible, by having the framework realized, as often done in many visual tools of this sort, by a user extend-able *meta-model* that defines the abstractions to be presented.

The SGT further includes wizards and easily configurable queries for user driven investigation of software artifacts. However, system architects still need help in identifying important areas to investigate. This work argues that such help can be automatically provided when the SGT tool is extended to include visual representation of valuable software metrics.

The context of this argument is as follows: As part of our ongoing effort to improve SGT and offer more useful abstractions, we evaluated the prospects of augmenting it with the ability to present software metrics. A user extending the meta-model has then to choose among the legions of metrics competing for screen real estate, precisely those metrics which are most useful, either in general or for his product.

This selection process will undoubtedly apply *validity* and *reliability* criteria: "What do these numbers mean?", "how do they reflect on quality?", "complexity?", are typical questions that one would ask when bombarded with a list of metric values. The quest of answering these validity questions has been a source for many challenges in software engineering research. One may also apply "stability" or "reliability" criteria to any given metric—after all, there is little point in presenting highly ephemeral information.

We argue that on top of these hard to apply considerations, and perhaps, even before applying these, the system architect should consider the scarcity of resources: limited screen space, limited spectrum of colors, and limited user attention. This is especially true when numerical values need

[*] Work done in part while author was with IBM Research–Haifa

to be presented; flooding the user with numbers is likely to be overwhelming and counter productive, even if these numbers are presented in less intimidating visual forms, e.g., color grades and shades or histograms.

SGT can tip the architect with its estimate on the *amount of information that each of the competing metrics provides*: the architect can then choose to present the metrics which are more informative than others, either in general, or for the product or set of products for which a meta-model is tailored.

The main contribution of this work is with a new criterion for evaluating metrics, which is based on a normalized version of Shannon's information theoretical content. We study this criterion in the context of a large software corpus, comprising close to eighty thousand Java classes and for a set of 36 software metrics. On course, we offer a taxonomy of software metrics, based on scope, range of values and other criteria. The software corpus was modeled using graph based techniques, which are widely utilized [3, 25, 26] in other computer algorithms.

Based on our empirical findings, we identify a candidate metric for visual presentation, describing the "centrality" of a module, as measured by a variant of Google's page-ranking algorithm [20]. Second to this comes the module size, as measured by the well known WMC metric [8].

There is an arguable merit to developers and maintainers in both metrics: the more central modules should be useful in identifying the software architecture, while the module size may be typical of complexity. Naturally, we stay short of the detailed study of the actual developers and maintainers's experience in using the studied metrics, and value they may draw from them. Such a study requires entirely different research methods, and could complement our work.

Similarly, the study of "contextual" information, such as interpretation of the meaning of identifiers, or the text found in program comments or other documentation[6, 10, 28], requires very different tools, and is beyond the scope of our work.

Outline. The remainder of this article is organized as follows. The data corpus and the way it was selected are described in Section 2.

Section 3 presents our metrics taxonomy, which is then used in Section 4 to present the metrics used in this study. Section 5 introduces a method of estimating how "informative" a metric is, along with other descriptive statistics of the numerical metrics in out suite. Section 6 repeats this analysis for Boolean metrics. Related work is the subject of Section 7, while Section 8 concludes.

2. Software Corpus

2.1 Artifacts

The software corpus used in our experiments comprised 19 software *artifacts*, all drawn from the *Qualitas Corpus* [27], a colossal collection of JAVA software that is being used extensively in many empirical software engineering studies[1].

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, `FreeCol` (a simulation game), `Antlr` (a framework for constructing compilers, interpreters, etc.) `hibernate` (a persistence framework), `holds` (a relational database engine), `jgraph` (a graph drawing package), `log4j` (the logging component of Apache), `struts` (the Apache framework for the creation of web applications), `weak` (data mining and machine learning software), `argouml` (an UML diagramming application), `hsqldb` (hyper SQL database engine), `jhotdraw` (java GUI framework for technical and structured graphics), `jung` (framework for modeling, analysis and visualization of graphs), and `proguard` (java shrinker, optimizer, obfuscator and preverifier).

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

Table 1. Size characteristics of the software corpus.

2.2 Versions

For each artifact, we analyzed a number of *versions* from the corpus. In total, our corpus comprised 95 versions.

The essential size characteristics of the corpus are summarized in Table 1. The corpus totaled some 78 thousands types, organized in 5,500 packages.

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 1 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 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,

[1] See `http://www.cs.auckland.ac.nz/~ewan/corpus/publications.html` for a partial list.

but they are robust to outliers, which are unavoidable with this great variety.

3. Taxonomy of Software Metrics

This section describes the 36 software metrics used in our experiments, and proposes a taxonomy of metrics of this sort.

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 $v \in G$.

3.1 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 (e.g., distinguishing between different kinds of dependencies among nodes). The suite includes 17 semantical metrics.

3.2 Metric directionality

The *dual* of a (topological) metric μ_G is a metric μ'_G, defined by $\mu'_G(v) \equiv \mu_{G'}(v)$ where G' is the graph obtained from G by inverting the direction of all edges in it. Thus, metrics μ_1 and μ_2 are duals if μ_1 computed in G is the same as μ_2 computed in G'. A metric is *undirected* if it is the dual of itself; it is otherwise *directed*. Our metrics suite includes 18 directional metrics and 18 unidirectional metrics.

3.3 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 [8] has a number of strictly local methods, including *Number of Children* (NOC) and *Coupling between Object Classes* (CBO), which is defined as the number of types whose methods may be invoked in response to call to the methods of a given type. 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 subcategory 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) [8], is an example of an internal metric.

A metric which is not local is *global*, e.g., the PageRank metric mentioned above is global.

3.4 Metric range

Our fourth 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.

4. Metrics Used in the Experiments

The literature defines hundreds if not thousands of code metrics.

Table 2 enumerates the metrics used in our experiments, classifying these according to this taxonomy.

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

Table 2. Metrics used in experiments and their categories

4.1 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 (so named after balloon types [5]) 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.

Following that, we have a group of micro-patterns markers [11]. For this work, we carried out measurements on seven of these.

4.2 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 [8], including the metrics described above, together with *Response For a Class* (RFC), which is the number of methods that can potentially be executed in response to an invocation of a method in the type.

The WMC metric was computed by using the total number of instructions in this method as method complexity. In addition to these basic metrics, we included a variant of DIT, *Number of Ancestors* (NOA) which seems appropriate for the inheritance structure of interfaces and classes in JAVA. Of this suite [8], the *Lack of Cohesion* (LOC) metric was not included in our study.

4.3 Plain Topological Metrics

The next local metric is *#Incoming*, which counts the number of immediate clients a type has. (Of course, this metric is related to *sink* and *wrapper* metrics.) 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, counting the number of types that a given type uses directly and indirectly; observe that *#Descendants* is identical to Page-Jones and Constantine's [21, 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.

4.4 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 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. A strongly connected component thus may be thought together of as *super module*. In our suite, *SCCSize* represents the size of this super module (i.e., the size of the strongly connected component) that a type belongs to. *#SCCIncoming* and *#SCCClients* are, respectively, the number of super-modules immediate and indirect clients that the super module serves. Their duals are *#SCCOutgoing* and *#SCCDescendants*.

4.5 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.

4.6 Other Metrics

In the last group of metrics in Table 2 we have *PageRank* and *Betweenness*, yet another measure of graph centrality [7]; 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] and in SA4j[3], 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.

5. Entropy and Information Density of Numerical Metrics

Having presented the metrics suite and the software corpus, we now present some summarizing statistics on the behavior of the metrics on the versions in the corpus. The amount of information that marker metrics may bring to the code inspector depends on their prevalence. We need more measures to appreciate the information that numerical metrics yield.

Range Our range categorization of metrics distinguished between continuous and discrete metrics. We shall use the letter $k = k(\mu, G)$ to denote the number of values that a metric μ assumes on a graph G. If k is small, then the metric cannot be too informative. But, too high a value of k is not very useful either. Consider PageRank, for example. The absolute values of PageRank are not very interesting. But, even an ordinal ranking can be confusing. When told that "the node corresponding to this class was placed in the 73.9^{th} percentile by PageRank", the third and even the second digit

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

[3] `http://www.alphaworks.ibm.com/tech/sa4j`

of accuracy in this statement will most likely be treated as noise.

Entropy Recall Shannon's definition of *entropy* [4], which gives a measure of the amount of information of a partition of a set of n elements into nonempty subsets sized $s_1, \ldots, s_k$,

$$H(n, s_1, \ldots, s_k) = -\sum_{i=0}^{k} s_i \lg\left(\frac{s_i}{n}\right). \qquad (1)$$

The above can be viewed as the number of bits required to represent the partition. Indeed, notice that the maximal value of the above, $n \lg n$, is achieved when $s_i = 1$ for $i = 1, \ldots, s_k$, while the minimum, 0, is obtained when the partition is into a single set.

Normalized Entropy Let

$$\widetilde{H}(n, s_1, \ldots, s_k) = \frac{H(n, s_1, \ldots, s_k)}{n} \qquad (2)$$

denote the *normalized entropy* of a partition, which can be thought of as the number of bits of information that the metric provides for each element participating in the partition. Then, in the case of a partition into singletons, the normalized entropy is $\lg n$, while in the case of a partition into two equally sized sets, the normalized entropy is 1.

The entropy of a metric μ is easily defined with (1), since μ partitions the nodes in the graph into equivalence classes, where the value of μ for nodes in each such class is the same. We can therefore use normalized entropy to compare the amount of information that two different metrics provide on the same software graph.

Note that a high value of $\widetilde{H}$ is not an end by itself. As we observed above, extra information can be overwhelming rather than meaningful.

Information Density. The normalized entropy measure is biased, at least in the case of continuous metrics, to favor larger graphs. As it turns out, the more nodes in the graph, the more values a continuous metric tends to have, and hence the greater the information it carries. To carry out the comparison between values on graphs of different sizes, we define the *information density* of a metric as

$$\alpha(\mu, G) = \alpha(n, s_1, \ldots, s_k) = \frac{H(n, s_1, \ldots, s_k)}{n \lg n}, \qquad (3)$$

where $s_1, \ldots, s_k$ are the sizes of the equivalence classes. Thus, the information density is 1 for metrics that provide maximal information, and 0 for metrics which are constant.

In a particular application of a metric to single software system, a small number of extreme values, e.g, low information density, may indicate a design problem in the system, e.g., when most of the nodes are in the same strongly connected component.

Table 3 summarizes the values of the above measures for our corpus and non-marker metrics. Note that the table includes also the duals of metrics when this dual makes sense.

Metric	k	$\widetilde{H}$	α (%)
DIT	5±1	1.6±0.3	19±3
NOA	9±3	2.1±0.3	25±3
NOC	11±4	0.7±0.2	8±3
CBO	38±12	4.1±0.2	48±5
RFC	88±31	5.4±0.3	64±5
WMC	224±111	7.0±0.5	81±5
#Incoming	33±13	3.1±0.2	37±7
#Clients	40±22	3.5±0.6	42±10
#Outgoing	27±9	3.3±0.2	38±4
#Descendants	34±20	3.8±0.7	45±10
#SCCIncoming	19±7	2.6±0.3	30±5
#SCCClients	37±21	3.0±0.6	38±8
#SCCOutgoing	19±7	2.6±0.3	30±5
#SCCDescendants	31±18	3.7±0.6	46±8
SCCSize	4±2	0.9±0.1	10±2
#DominatedBy	4±1	0.9±0.2	11±3
#DominatedBy'	4±1	1.0±0.1	12±2
#DominatorHeight	4±1	0.6±0.1	7±2
#DominatorHeight'	4±1	0.6±0.1	6±1
#DominatorWeight	9±2	0.7±0.1	8±2
#DominatorWeight'	10±2	0.7±0.1	8±2
PageRank	229±129	6.0±1.0	69±11
PageRank'	300±175	7.8±1.0	88±3
Betweeness	92±57	3.0±0.5	33±4
Betweeness'	102±65	3.0±0.5	35±4
Belonging	79±40	4.4±0.5	47±5

Table 3. Median values of range, normalized entropy, and information density of numerical metrics applied to versions in the corpus.

We draw attention of the reader to the following observations:

- *Variety in range between metrics.* The measure k varies greatly between the various metrics.
- *Low information density in many small ranged metrics.* For example, almost all the metrics in the dominators tree example, yield about four values, but these four values are not distributed equally, as the $\widetilde{H}$ column tells. An equal distribution would have given 2 bits of information per measured type. However, in this group, values are less than one.
- *High information density in large ranged metrics.* PageRank and WMC exhibit the highest information density amongst the metrics.

- *Dominator tree metrics have few values.* Metrics in the dominator tree group have very few values. This indicates that the dominator tree is almost degenerate. The height of this tree is typically four, and the maximal number of nodes dominated by a node is about nine or ten.

6. Marker Metrics

We now turn to the analysis of the marker metrics. Table 4 gives the 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

Table 4. Essential statistics of the prevalence of the marker metrics in the suite.

It is easy to pick out interesting bits of information from the table, e.g., about 4% of all types are **`abstract`**, while another 15% being **`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 almost half of the classes; the **`sink`** and **`source`** markers. The prevalence of sinks is low, but still could be very telling 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 dataset.

However, it is even easier to get lost with all these numbers and with their meaning and interpretation. In the context of a design of a meta-CASE tool, such as SGT, it is much more useful to examine properties common to all metrics. In comparing the $\min$ and $\max$ columns we can quickly observe that there is a great variety in the prevalence of metrics (and accordingly, in the amount of information it carries), depending on the examined software artifact/version. This observation is also supported by the comparison of the standard deviation (the $\pm$ adornment of the "mean" column), with the mean: we see that the standard-deviation is almost always greater the mean. Thus, a decision of presenting a marker metric should be made application- or at least application-domain specific.

The median column is also interesting: in approximating the error of the median, we have used the M.A.D (median absolute deviation from the median), which is more robust to outliers than the standard-deviation. Still, M.A.D statistics is typically large in comparison to the median (the only notable exceptions being the "wrapper" and "stateless" markers).

This great variety strengthens our conclusion that the presentation of marker metrics must be user configurable.

The essential statistics of the normalized entropy of the marker metrics in the suite is presented in Table 5. We use normalized entropy instead of information density since markers are constrained, by definition, to two values; a division by an extra $\lg n$ factor will bias the results in favor of smaller software artifacts.

Metric	Mean	Median
final	0.47 ± 0.36	0.38 ± 0.33
abstract	0.26 ± 0.10	0.25 ± 0.09
interface	0.45 ± 0.17	0.41 ± 0.17
sink	0.08 ± 0.10	0.06 ± 0.06
source	0.74 ± 0.28	0.87 ± 0.12
balloon	0.44 ± 0.23	0.37 ± 0.17
wrapper	0.80 ± 0.10	0.79 ± 0.05
pure	0.41 ± 0.17	0.41 ± 0.12
pool	0.10 ± 0.07	0.08 ± 0.04
designator	0.03 ± 0.04	0.02 ± 0.02
function pointer	0.01 ± 0.03	0.00 ± 0.00
stateless	0.84 ± 0.12	0.87 ± 0.07
sampler	0.07 ± 0.05	0.07 ± 0.02
canopy	0.62 ± 0.20	0.63 ± 0.19

Table 5. Normalized entropy of marker metrics.

Examining the table, we see that many marker metrics carry minimal information, e.g., "sink" offers less than a tenth of a bit of information. User interface design should give such metrics special attention: if these low information metrics are to be presented, this must be in the form that does not clutter the main display, e.g., since the vast majority of types are not "designator"s, then only types which are "designator"s, should be marked as such—no special flagging should be made of types which do not meet the criteria of this micro-pattern.

7. Related Work

A measure of the relative importance of components within the software structure was examined in [18]. The authors suggested to use CODERANK, the software equivalent of Google's well known PAGERANK [20] method for ranking web pages, metric to indicate how important a specific component is based on its coupling to the rest of the system. In an earlier work [14], for the same purpose the authors suggested to use a similar metric called COMPONENT RANK. The main difference between these metrics is that the CODERANK is computed based on the weighted graph that represents various usage relations between the components and the number of time each usage occurs. This research is consistent with our finding that PAGERANK is an informative metric.

Lorenz et al. [17] recommend using a wide range of metrics to test the quality of models, classes and methods.

Various metrics related to coupling, inheritance and size of classes and methods play the major role in deducting the quality of the software. Our work may help decide which metrics out of this wide range should be presented to the architect as the most important to look at.

Lajios et al. [15] investigated the correlation of various software metrics to the defect found in software modules and proposed an approach to determine a sets of metrics for quality assessment of complex software systems. First they calculated various quantitative, complexities, coupling and other metrics at the class level for several similar projects using different open source tools. Then they found the correlation of these metrics to the history of bugs using machine learning techniques. They found that although some of the metrics are more suitable for the assessment of software quality, these metrics differ between the analyzed projects even though their natures are similar. They also discovered that 5 out of 11 metrics were irrelevant for the analyzed systems. This research completes ours in the attempt to find which metrics are informative and which are irrelevant.

Ordonez et al. [19] examined various metrics used in software industry to measure code size and design complexity. They mentioned that NASA used the first five metrics presented in [8] in the tool they developed for analyzing source code with respect to its architecture. The author's analysis was focused on how reliable are specific software modules with respect to their maintainability and the probability of causing defects. The information those metrics carry was not covered and might be useful for this work as well.

8. Conclusions

We presented a metrics suite comprising 36 code metrics drawn from various independent sources, and offered a taxonomy in which these, and many others can be organized. Our taxonomy of metrics included a distinction between semantical and topological metrics, a breakdown by directionality, and range of values yielded by the metric.

We observed that numerical valued metrics present a challenge for visual CASE tools such as SGT. Flooding the user with numbers is likely to be overwhelming and counter productive. Even though there are means for presenting numbers in less intimidating visual forms, e.g., color grades and shades, or as histograms, the design of SGT (and other SGT like tools) meta-model should try to minimize the use of resources such as screen real-estate, color spectrum and most importantly, user attention.

Our study proposed the information density property of a software metric as a criterion for selecting candidates competing on these resources. A very noteworthy candidate for visual presentation is the "centrality" of a module, as measured by what we called, PageRank', which is nothing but the application of Google's famous and now classical page-ranking algorithm to the graph of dependency among software modules (more precisely, to the graph in which the inverse of use-edges are used).

The second informative candidate is the module size, as measured by the well known WMC metric. Of course, other metrics that we did not investigate could make even better candidates.

Boolean metrics were discussed as well, with the main observation being that the decision of their use is very much application dependent.

References

[1] Rigi: A visual tool for understanding legacy systems. http://www.rigi.csc.uvic.ca/, 2011. URL `http://www.rigi.csc.uvic.ca/`.

[2] Source2value. http://www.omnext.net/source2value/, 2011. URL `http://www.omnext.net/source2value/`.

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

[4] O. Aktunc. *Entropy-Based Measurement for Software: An Entropy-Based Measurement Framework for Component-Based Hierarchical Systems*. VDM Verlag, Saarbrücken, Germany, Germany, 2008. ISBN 3639087259, 9783639087253.

[5] 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)*, pages 32–59, Jyväskylä, Finland, June 9-13 1997. Springer-Verlag. ISBN 3-540-63089-9.

[6] R. F. Andrian Marcus, Denys Poshyvanyk. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Trans. Software Eng.*, 34(2):287–300, 2008.

[7] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering.*, 20(6):476–493, 1994.

[9] Y. Dajsuren, M. Goldstein, and D. Moshkovich. Modernizing legacy software using a system grokking technology. In *26th IEEE International Conference on Software Maintenance (ICSM '10)*, pages 1–7, Timisoara, Romania, Sept. 2010. IEEE Computer Society.

[10] M. Gethers and D. Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *ICSM '10*, pages 1–10, 2010.

[11] J. Y. Gil and I. Maman. Micro patterns in java code. *SIGPLAN Not.*, 40(10):97–116, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1103845.1094819.

[12] M. Goldstein and D. Moshkovich. System grokking - a novel approach for software understanding, validation, and evolution. In *Proc. of the 7th International Conference on Next Generation Information Technologies and Systems, (NGITS 2009)*, pages 38–49, Haifa, Israel, June 16-18 2009. Springer-Verlag. ISBN 978-3-642-04940-8.

[13] IBM. Rational Rhapsody. http://www-01.ibm.com/software/awdtools/rhapsody/, 2011.

[14] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: Relative significance rank for software component search. In *Proc. of the 25th International Conference on Software Engineering, (ICSE 2003)*, pages 14–24, Portland, Oregon, USA, May 3-10 2003. IEEE Computer Society.

[15] G. Lajios. Software metrics suites for project landscapes. In *Proc. of the 2009 European Conference on Software Maintenance and Reengineering.*, pages 317–318, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[16] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *Proc. of the 2008 international symposium on Software testing and analysis, (ISSTA'08)*, pages 131–142, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0.

[17] M. Lorenz and J. Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994. ISBN 0-13-179292-X.

[18] 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)*, pages 369–378, Sydney, Australia, Apr. 18-21 2006. IEEE Computer Society. ISBN 0-7695-2551-2.

[19] 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)*, pages 453–458, Las Vegas, Nevada, USA, Apr. 7-8 2008. IEEE Computer Society. ISBN 978-0-7695-3099-4.

[20] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, Nov. 1999. URL `http://ilpubs.stanford.edu:8090/422/`.

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

[22] D. Redmond-Pyle. Software development methods and tools: some trends and issues. *Software Engineering Journal*, 11(2): 99–103, 1996.

[23] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 167–176, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094824. URL `http://doi.acm.org/10.1145/1094811.1094824`.

[24] R. S. Sangwan, P. Vercellone-Smith, and P. A. Laplante. Structural epochs in the complexity of software over time. *IEEE Software*, 25(4):66–73, 2008. ISSN 0740-7459.

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

[26] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[27] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.

[28] B. Újházi, R. Ferenc, D. Poshyvanyk, and T. Gyimóthy. New conceptual coupling and cohesion metrics for object-oriented systems. In *10^{th} IEEE Working Conference on Source Code Analysis and Manipulation (SCAM'10)*, Los Alamitos, CA, USA, 2010.

[29] R. Wuyts, H. M. Kienle, K. Mens, M. van den Brand, and A. Kuhn. Academic software development tools and techniques. In *Proc. of the 22nd European Conference on Object-Oriented Programming, (ECOOP 2008)*, volume 5475, pages 87–103, Paphos, Cyprus, July 7-11 2008. Springer.