A metric framework for object-oriented development

Tanit Talbi Bertrand Meyer Emmanuel Stapf
Interactive Software Engineering Inc.
ISE Building, 360 Storke Road, Santa Barbara CA 93117 USA
http://www.eiffel.com
tanit@eiffel.com

Abstract

Metrics (quantitative estimates of product and project properties) can, if defined from sound engineering principles, be a precious tool for both project management and software development. We have recently developed an extensive set of metrics facilities for the EiffelStudio development environment. We will describe the principles on which it is based, the facilities it provides, and how to use them. The metrics workbench is closely integrated with the rest of the environment. Among other capabilities, it allows users to: apply predefined metrics to components of a system at various levels (feature, class, cluster, entire system); define new metrics, through mathematical formulae or boolean selection, and apply them to projects; store measurement results, as well as metric definitions, into an XML archive that can be stored locally or made available on the Web for future reference; compare the measurements on a system to those on record locally or on a Web site. ISE has released on its own site an archive recording the metric properties of its basic libraries, available to any other project for comparison.

1. Introduction

Although one should resist the tendency to believe numbers just because they are numbers ("lies, damn lies, and metrics"), there is no reason why software engineering shouldn't be able to reap at least some of the benefits that precise quantification has brought to other engineering fields. Software metrics, which we may define as quantitative estimates of product and project properties, can indeed help both managers and developers.

Object-oriented development, with the rich software structures that it induces, is a particularly amenable to metric analysis. Even when some of the measures do not seem to bring much by themselves, *comparing* them to those of other projects may point to important properties of the software.

One of the innovations of the EiffelStudio, the development workbench of the new ISE Eiffel 5 environment, is a set of metric facilities enabling developers and managers to obtain quantitative information about software systems. Particularly important is the possibility of comparing any measure to some in record locally or through the Web; ISE is releasing results for its libraries and other products for public access at **http://metrics.eiffel.com.**

This article describes the principles behind the metric tool as well as its practical use.

2. Principles

2.1 General approach

We started out with a skeptical attitude towards software metrics, and a determination to avoid producing numbers that might impress a manager or customer but would lack a proper

scientific justification. We felt that if we had to err it should be on the side of intellectual prudence in the face of ever-present temptation to throw numbers at problems and people. Since we were building a practical tool, not a new metric theory, we resolved:

- Never to implement a metric facility unless we could convince ourselves that it was directly and undeniably related to some relevant attribute of a software product or process.
- Never to succumb to arguments of the style "maybe we don't see a need for this, but it's simple to add to what we have already implemented, and someone might make sense of it".

Instead the general attitude has been "if it turns out to be necessary we can always add it later". This is made easier, of course, by the flexibility of O-O architectures as supported by Eiffel (the environment is written in Eiffel and bootstrapped).

A metric framework should satisfy the following properties:

- **Coverage:** include a definition of what is being measured, sufficient to enable repetition of the measurements.
- **Trustworthiness**: include an estimate of how much the results can be believed, in particular of their precision (expected variations in case of repetition).
- **Relevance**: specify interesting properties of software products or processes on which the measurement may provide insight.
- **Theory**: include arguments backing the statement of relevance.

The first step was to define precisely the domain of discourse.

2.2 Metrics, measures, and metric theories

One shouldn't confuse metrics with measures. A metric is a quantitative property of software products (product metrics) or processes (process metrics) whose values are numbers — either integer or real in our current framework). A measure is the value of a metric for a certain product or process.

For example, we can evaluate the metric "number of classes in the system", called just *Classes*, by counting the classes in our system. This yields a measure.

In software we may distinguish between *product* metrics, which measure properties of the elements being turned out (code, designs, documentation, bug reports…) and *process* metrics, which measure properties of the process whereby they are being turned out (salaries, expenses, time spent, delays…). The current metric facilities of EiffelStudio are primarily product-oriented, although; a few metrics, such as "number of compilations", are process metrics. To add product metrics requires interfacing with project management tool; this is a desirable development for the future.

Any metric should be *relevant* related to some interesting property of the processes or products being measured: cost, estimated number of bugs, ease of maintenance… A *metric* theory is a set of metric definitions accompanied with a set of convincing arguments to show that the metrics are relevant. Neither EiffelStudio nor this article provides a metric theory; our purpose is simply to provide the basic tools that enable the development and application of good metric theories.

2.3 Units

The numbers yielded by measures are meaningless unless we describe what they refer to. Every metric is relative to a certain unit, specified as part of its definition. For example the unit for a metric that counts classes, such as Classes, is called CLASS.

The environment provides a set of predefined units. Some simply serve to count occurrences of certain construct specimens in the software; examples include CLASS,

CLUSTER, FEATURE, LINE, ... The metric RATIO is used to describe metrics whose values are division, for example “average number of classes per cluster”, obtained by dividing the number of classes by the number of classes.

We considered the possibility of developing a full calculus of units, similar to what exists in some of the natural sciences, with units such as cm/h and g/cm2, and rules stating for example that the produce of the last two is $g/(cm \times h)$. We rejected this idea because it seemed like overkill; in particular we couldn’t immediately see the need for multiplications, or for divisions of divisions. This is why we stop at a single division and have just one unit, *RATIO*, for this case; it’s a consequence of the concern for simplicity and intellectual prudence cited above. If future experience shows that this policy was too restrictive, we may reconsider the decision and introduce a unit calculus.

2.4 Scope types and scopes

Any metric applies over one or more scope types. A scope type is a type of product or process over which the metric is measured; for product metric, examples include “feature” (meaning that we will compute a metric over a single feature), “class”, “cluster”, “system” , “set of systems”. These obey an order relation corresponding to the containment order of the corresponding software elements: a feature belongs to a class, a class to a cluster and so on. All except the last one are currently available.

A scope is a particular instance of a scope type. For example a given cluster is an instance of the scope type “cluster”.

To compute a measure is to apply a certain metric over a certain scope of an applicable scope type. For example we may compute the value of the metric Classes over a certain system.

3. Metric classification

The EiffelStudio metric framework provides a number of predefined metrics but also enables users to define their own metrics in terms of the predefined ones. Figure 1 shows the overall taxonomy,

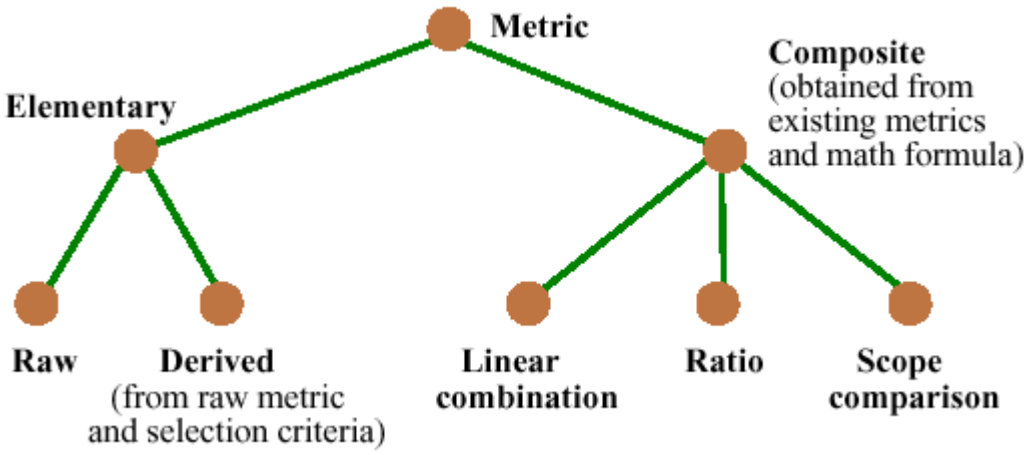

Figure1. Metric hierarchy

3.1 Composite and elementary metrics

Metrics are divided into elementary and composite.

An elementary metric measures the number of occurrences of a certain pattern in the product or process. An example is the number of precondition clauses in a class.

A composite metric, defined by a user of the environment, applies a mathematical or logical formula involving other metrics (elementary or previously defined metrics). Composite metrics are discussed below.

3.2 Raw and derived metrics; selection criteria

Among elementary metrics, we make a further distinction between raw and derived:

- Raw metrics are simple counts, built-in into the environment, of occurrences of certain basic elements. For example *Classes*, the number of classes, is raw.
- It is often useful to define a new metric by subjecting a raw metric to one or more *selection criteria*. For example a class may be either deferred (abstract) or effective (concrete, i.e. fully implemented). This is a selection criterion. Separately, a class may have an invariant, or not; this is another selection criterion, *Invariant_equipped*. You might want to know the number of classes that are deferred and have no invariant clause; if so, you may define a derived metric by submitting the raw metric *Classes* to both of these criteria, connected by an "and" combinatory.

The precise definition of selection criterion for a raw metric is: a property with a fixed set of possible values (two or more) characterizing the patterns being counted by the metric.

The reason for considering selection criteria and derived metrics is clear: without these notions, the environment would need to have predefined (raw) metrics including all possible combinations, such as "deferred and no invariant". This would quickly grow out of hand.

Figure 2. Interface for defining derived metrics

Figure 2 shows an example, which should be self-explanatory, of the interface for defining a new derived metric. Note the large number of selection criteria applicable to the raw metric Features (number of features), reflecting the many angles under which features may be classified.

3.3 Composite metrics

A composite metrics applies one or more mathematical operators to a set of metrics, themselves either elementary (raw or derived) or already composite.

Although we considered allowing arbitrary mathematical combinations, we settled — again in application of the Principle of Prudence — to limit the possibilities to exclude multiplication of metrics and other combinations for which we could find no clear arguments of relevance, and to include the following three kinds only:

- Linear metrics: metrics of the form $\sum k_i \,.\, m_i$, where the k_i are real values and the m_i existing metrics (either elementary or basic) with the same unit, other than *RATIO*. (It would be improper to add two *RATIO* since they might be ratios of incompatible things.) Figure 3 shows the interface for defining a new linear metric.

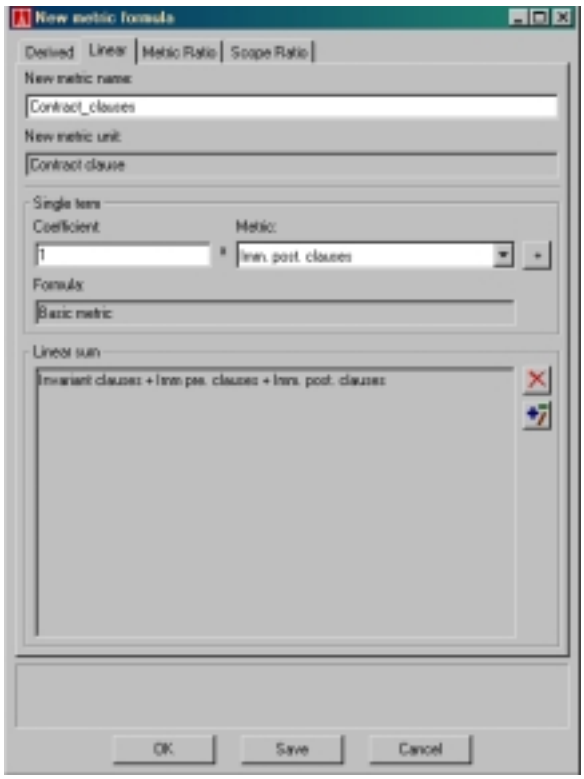

Figure 3. Interface for defining linear metrics

- Ratio metrics: metrics of the form $m_1 \,/\, m_2$ where the m_i are two previously defined metrics, not necessarily with the same unit, neither of which a *RATIO* (again because *RATIO* is a catch-all category for all divisions, so we can't divide further without courting incoherence). The resulting unit is *RATIO*. Figure 4 shows the interface for defining a new ratio unit.

Figure 4. Interface for defining ratio metrics

- Scope comparison metrics: metrics that measure the ratio of the value of a given non-ratio metric over two different scope types. For example by choosing the metric *Classes* and the scope types "cluster" and "system" we can measure the proportion of classes in a system that belong to the current cluster. Figure 5 shows the interface for defining a new scope comparison metric.

Figure 5. Interface for defining scope comparison metrics

4. Measurement

Not all metrics may be applied to all scopes. For example it doesn't make sense to compute the number of attributes in a feature, since the smallest construct in which attributes appear is a class, bigger than a feature.

To formalize this notion we say that each raw metric has one or more basic scope types, the type of scopes on which the environment has built-in mechanisms to compute it. The list of basic scope types is part of the metric's definition. Then the rule to compute the metric on any scope of scope type st is as follows:

- 1. If *st* is one of the metric's basic scope types, apply the environment's built-in mechanism to determine the result.
- 2. If *st* is smaller than the smallest of the metric's basic scope type, the result is zero by convention.
- 3. Otherwise, the computation will add the measures made on the constituent scopes, applying the rule recursively.

This rule applies to raw metrics; it immediately generalizes to derived and composite metrics.

There is a small subtlety in the rule, explaining also why we introduced the possibility of several basic scope types rather than just one. The following example justifies this convention and the rule. Consider the metric "number of source lines". The scope type just above "feature" is "class"; in other words, a class includes features. Each feature has a certain number of lines; each class also has a certain number of lines. But we can't obtain the number of lines of a class by adding the number of lines of its features per clause 3 of the rule, because a class may also contain lines that are not in features, for example invariant lines. For that reason the definition of the metric includes both "feature" and "class" as basic scope types, so that the environment has built-in rules to compute the number of lines in these two cases. For other cases, such as requesting the number of lines in a cluster, it will just add the results for all immediate constituents — the cluster's classes — as per clause 3.

5. Performing measures

Once you have defined metrics, you will want to compute corresponding measures on parts of your software. Figure 6 shows the interface for performing a measure over a certain scope.

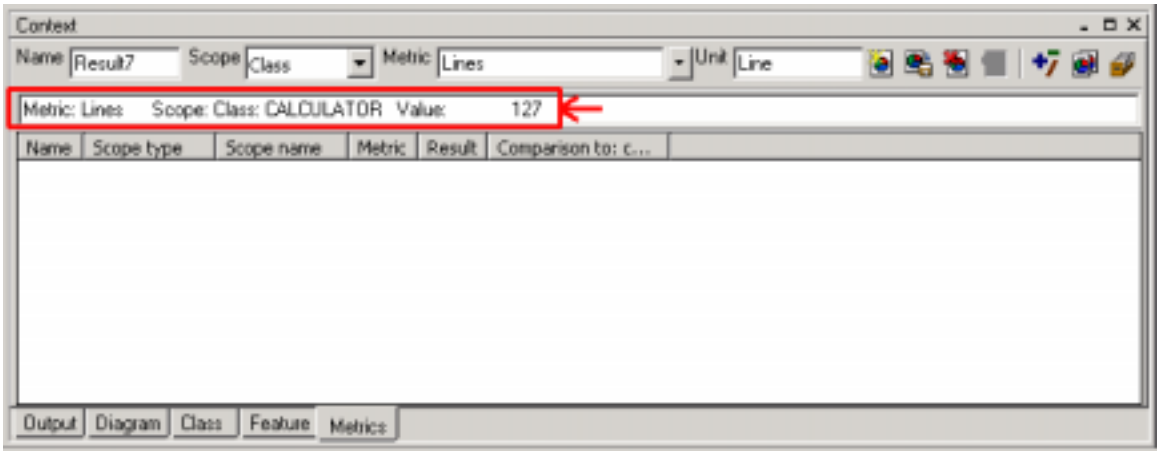

Figure 6: Evaluation of a metric over a scope.

5. Storing and importing measures and metrics

Figure 7 shows a project with a number of measures previously performed. The last column is a comparison to a reference archive and is explained in the next section.

By clicking on the button, you may save both the current metrics definition and the measures for later use. The storage uses an XML format. You may also import metric definitions from another project into your current project through the button.

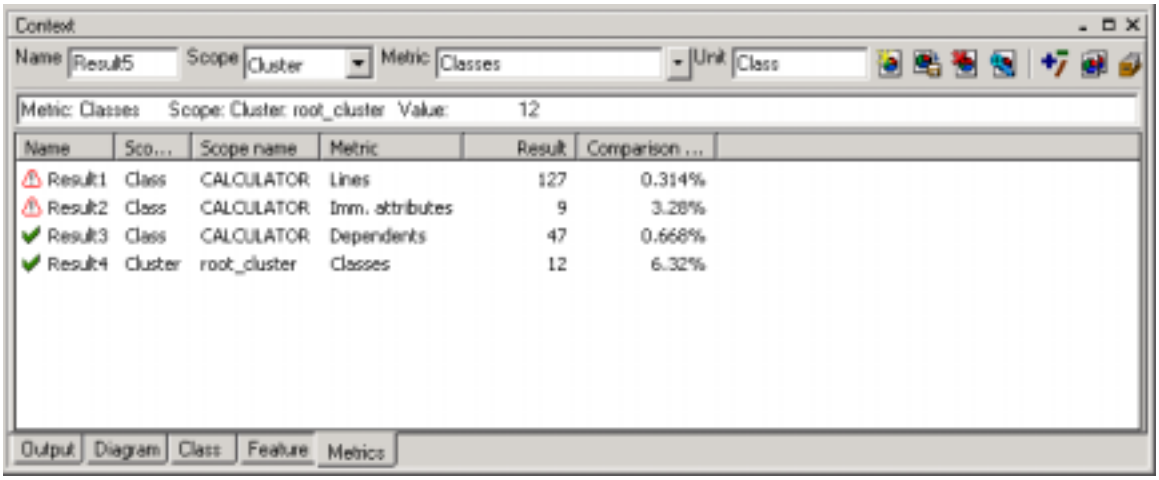

Figure 7: Saved measures.

6. Measurement archives

What does a measure mean? You don't necessarily know in the absolute, but you might want to compare your results to those of other projects. For example, if you have measured the average number of invariant or other contract clauses in your system, you might be curious to know how this compares to ISE's EiffelBase library.

The notion of metric archive addresses this need. We expect it to be one of the most attractive uses of the tools. You may:

- As noted above, archive all current measures into a file, called a **measurement archive**.
- Make this measurement archive available in a shared directory, or as a **URL on the Internet**.
- At any time in a project, select any measurement archive, local or URL, as the **reference archive**; in that case all measures that you perform will be compared to those of the reference archive. You may select various comparison formats: percentage (the default, used in figure 7), difference percentage, plain value.

ISE has established a new Web site, http://metrics.eiffel.com, as a publicly available reference for metric collections on ISE's own libraries (EiffelBase, EiffelVision, …) and products. This provides an invaluable source of comparisons for other projects within and without ISE.

6. Conclusion

The metric workbench of ISE Eiffel 5 provides a set of basic measurement facilities. It does not suggest a particular metric theory, but we hope it has the tools through which its users, by following a particular theory and implementing it through the metric definition facilities, can gain a deeper understanding of their software products and processes, and as a result improve them.

Bibliography

Although not cited in the text, the following references provide some of the background for this work.

[1] Bogdan Durnota and Christine Mingins: Tree-based coherence metrics in object-oriented design. In TOOLS 9, *Technology of Object-Oriented Languages and Systems*, eds. C. Mingins, B. Haebich, J. Potter, and B. Meyer, Prentice-Hall, Englewood Cliffs (N.J.), 1992, pages 489–504.

[2] Bertrand Meyer: *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice Hall, 1995.

[2] Christine Mingins, Bogdan Durnota and Glen Smith: *Collecting software metrics data for the Eiffel class hierarchy*, in TOOLS 15, *Technology of Object-Oriented Languages and Systems*, eds. C. Mingins and B. Meyer, Prentice Hall, Englewood Cliffs (N.J.), 1993, pages 427-435.

Acknowledgment

We are grateful to Christine Mingins from Monash University and Xavier Rousselot from ISE for important comments during the development of the metric tools.