A metrics framework

Bertrand Meyer

Draft 11, February 2001. *Copyright ISE, 2001.*

This document is an internal ISE working paper. Although it served as a guide in the implementation of the metrics facilities of ISE Eiffel 5.5, it is a discussion paper and not the description of any specific tool. The paper is only a very preliminary draft and we expect that it will be extensively revised as we expand our reflection on software metrics.

Abstract

A discussion of the role and scope of software metrics, clarifying their purpose and introducing a number of definitions, with the goal of specifying the proper metrics tools for an O-O environment.

1 CONTEXT

One of the innovations of the new ISE Eiffel 5 environment is a set of metric facilities enabling developers and managers to obtain quantitative information about software systems and the process of producing them. These tools will let you, among other applications:

- Find out basic quantitative properties of a system, such as the number of classes, the number of features, how many times the system has been compiled and many others.
- Define new properties as mathematical combinations of the basic ones, and compute the resulting values, to get the answer to such questions as "what percentage of my system's routines have a header comment?" or "what is the average number of features, of lines, of invariant clauses per class?".
- Vary the *scope* of such measurements, so that the measured result may cover a single feature, a class, a cluster of classes, an entire system, a set of reference systems, or all the systems built by a group, a company or a group of companies having submitted their data.

- Compare the result of the same metric over several of these scopes, to see for example how a project differs from those on record, or whether some parts of a project depart from the norms applied in others.

The environment will support such measurements through a simple graphical interface, closely integrated with the rest of the Eiffel Explorer.

Users of the environment can start applying these facilities to their systems without learning any theory. Metrics, however, are notoriously subject to abuse; we must be wary of the temptation to revere numbers just because they are numbers ("*lies, damn lies and metrics*"). This note presents a simple approach to software metrics designed to establish a clear and sound basis, so that we know what we are measuring, why we are measuring it, and how much value we may attach to the results.

2 REQUIREMENTS

All engineering disciplines other than software rely on quantitative approaches to design their products, organize the production process, and assess the results. Software engineering too can benefit from measuring relevant properties of its products and processes.

Measurements are not an end in themselves; they must be related to broader goals of software engineering:.

Purpose of software measurements

The purpose of software measurements is to provide quantitative *assessments* or *predictions* of properties of software products and processes.

For any numerical measure that we propose we must have precise information on four properties: coverage, trustworthiness, relevance, theory.

Desirable features of software measurements

Any quantitative report on software products or processes should satisfy the following properties:

- **Coverage**: include a definition of what is being measured, sufficient to enable repetition of the measurements.
- **Trustworthiness**: include a 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.

3 DEFINITIONS

The first task is to define precisely the concepts involved. For example, a metric is not the same thing as a measure.

3.1 Attributes, metrics and measures

The most general notion is “attribute”:

Attribute

An **attribute** is a property, qualitative or quantitative, of software products or processes.

We may distinguish between the product and process cases:

Product attribute, process attribute

A **product attribute** is an attribute that characterizes a software product or set of products.

A **process attribute** is an attribute that characterizes a software-related process, such as development, maintenance, documentation, management, or multiple instances of such a process.

Examples of attributes include reliability (a product attribute, non-quantitative) and total project cost (process, quantitative).

A metric is simply a quantitative attribute:

Metric

A **metric** is an attribute whose values are numbers (integers or reals), expressed relative to a certain **unit** specified as part of the metric’s definition.

Examples of metrics include the number of source lines of a program (product) and the total cost of a project (process).

Attributes other than metrics will be called “qualitative”:

Qualitative attribute

A **qualitative attribute** is an attribute other than a metric.

An example of qualitative attribute is the reliability of a software product.

The “process” vs. “product” distinction carries over to metrics:

Product metric, process metric

A metric is a **product metric** if it is a product attribute, a **process metric** if it is a process attribute.

"Relevance", as defined in the previous section, suggest that the purpose of metrics is to help us gain information about attributes that are of direct interest to us. Often these will be qualitative; for example we may want to estimate the reliability of our software. Metrics provide us with numerical values that can serve to assess or predict such attributes.

Applying a metric will give us measures:

Measure

A **measure** is the value of a metric for a certain process or product.

3.2 Units

Metrics will be expressed in units, such as *LINE* (numbers of source lines) or *FEATURE* (number of features):

Unit rule

The definition of any metric must specify an associated unit.

Different metrics may be marked with the same unit; for example metrics such as number of non-comment lines and number of comment lines may both be marked with the unit *LINE*.

Whenever we need to express ratios of one measurement to another, as in computing the average number of features per class by dividing the number of features by the number of classes, we will use a specific unit, *RATIO*:

"Ratio" unit

The name *RATIO* denotes a predefined unit, whose values are arbitrary real numbers, expressed either as numbers or in percentage style (as in *35.3%*).

As a result of this choice, we will consider every division to yield a result of unit, *RATIO*.

In the future, it might be desirable to include more specific units, such as "features per class", and an associated calculus of units (as in the physical sciences, where multiplying a quantity expressed in g / cm^2 by one in cm / g yields a result in *cm*). This has not appeared necessary for the typical uses of division envisioned in this note, answering such questions as "What percentage of routines have a header comment?" and all expressed as simply a *RATIO*.

3.3 *Metric frameworks and theories*

You will want to rely not on a single metric but on a combination of metrics:

Metric framework

A **metric framework** is a set of definitions of metrics.

Any metric work should be backed by a theory:

Metric theory

A **metric theory** is a combination of

- A metric framework.
- A set of definitions of attributes (qualitative or metric).
- A mapping from the framework to the set of attributes, representing the hypothesis that each metric is a good predictor of the associated attribute.

This note does not introduce a metric theory, but it defines a metric framework by introducing a number of metrics.

3.4 *Elementary and composite metrics*

Some of our metrics will be *elementary* and some *composite*. An elementary metric is measured directly from the product or project record:

Elementary metric

An **elementary product metric** is a product metric whose values (integers) indicate the number of occurrences of a certain pattern in a product.

An **elementary process metric** is a process metric whose values reflect measurements drawn directly from project records.

An example of elementary product metric is the number of source lines. An example of elementary process metric is the number of incremental compilations of a system.

From these elementary metrics we may define composite ones:

Composite metric

A **composite metric** is a metric whose values are defined by a mathematical formula involving other metrics (elementary, or previously defined composite metrics).

A later section will introduce a number of operations for defining composite metrics out of elementary ones.

→ *"DEFINING A COMPOSITE METRIC", 4, page 10*.

Again we may distinguish between product and process:

Composite product metric, composite process metric

A **composite product metric** is a composite metric defined entirely in terms of product metrics.

A **composite process metric** is a composite metric whose definition involves one or more process metrics.

By convention, this definition treats as process metric a composite metric involving both product and process components.

The classification introduced for metrics extends to measures, so that we may talk about an elementary product measure, a composite process measure and so on.

3.5 Raw metrics and selection criteria

Elementary metrics measure patterns whose occurrences (in a product or process) can be counted. We need to decompose this notion further to avoid an explosion of the number of elementary metrics. For example the features of a class can be classified along several lines:

- Some are attributes (object fields), others are routines (algorithms).
- Some are inherited from a parent, others immediate (defined in the class).
- Some are exported, others secret

and so on. Many combinations of these properties may be worth counting on their own: you may ask for the number of secret attributes, of exported inherited features and so on. But if we define all of these metrics independently, we will soon have too many elementary metrics, while still failing to satisfy user needs if we have omitted a particular combination.

To avoid this we must identify a subset of elementary metrics as **raw metrics**: metrics whose results are counted directly. “Number of features in a class” is a raw metric. From a raw metric, we may then derive other elementary metrics by applying **selection criteria** such as “Is this feature an attribute?” or “Is this feature exported?”. Here are the definitions:.

Raw metric, selection criterion, derived metric

An elementary metric is either **raw** or **derived**.

A **raw metric** is an elementary metric not defined in terms of another metric.

A **selection criterion** for a raw metric is a property, with a fixed set of possible values (two or more), characterizing the patterns or events being counted by the metric.

A **derived metric** is an elementary metric defined from a raw metric by counting only the patterns satisfying a certain combination of its selection criteria.

To define a derived metric, we start from a raw metric, for example "number of features", and combine some of the associated selection criteria. The combination may be:

- An "or": for example, count all the features that are attributes or exported (or both).
- An "and": count all the features that are both attributes and exported.

Our metric framework does not provide more general boolean combinations, although they would be easy to add if necessary.

A selection criterion may have more than two possible values. Most, as in these examples, have just two: a feature is either an attribute or a routine; it is either exported or secret.

The definition states that every derived metric is derived from a certain raw metric. This provides only for two levels of elementary metrics, raw and derived; you may not, from an existing derived metric, derive new ones, since you may only associate selection criteria with raw metrics.

As a general guideline, we will try to keep the selection criteria for a given raw metric independent. For example, to distinguish the classes that inherit from a given class we will use the following selection criteria:

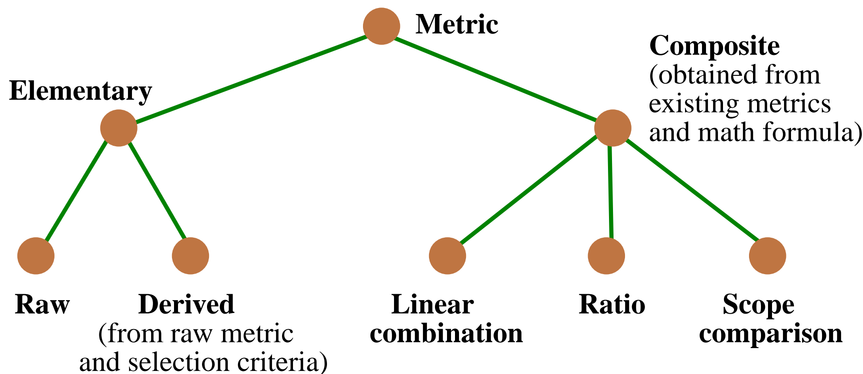
- *Heirs*: count the number of direct heirs of a class.
- *Indirect_heirs*: count the number of indirect heirs.
- *Self*: count only the class itself.

Other interesting notions are "proper descendant", covering direct and indirect heirs, and "descendant", covering proper descendants and descendants. We do not introduce these as separate criteria, since this would result in a non-independent set of criteria: every proper descendant is a descendant, every heir (direct or indirect) is a proper descendant. Instead we limit ourselves to the three criteria listed above. If you need a finer decomposition, it is easy, with the techniques discussed below, to define *Proper_descendants* and *Descendants* as composite metrics expressed in terms of the elementary metrics *Heirs*, *Indirect_heirs* and *Self*.

→ "DEFINING A COMPOSITE METRIC", 4, page 10.

This criterion independence principle is not absolute, however, and in some cases we may find it clearer to define a new criterion even though its value is entirely determined by certain values of another. For example one criterion on features determines whether it is an *attribute* or a *routine*, another whether it is *deferred* or *effective*. An attribute is always effective, so the two criteria are not independent. Even though we could remove the dependency by using a single criterion with three values (attribute, effective routine, deferred routine), it is more convenient to keep two criteria. Of course the underlying counting mechanisms must make sure never to count an element twice even in an "or" query, as when a user asks for the number of features that are attributes or effective.

The following diagram will help remember the various kinds of metric, as used in the rest of the discussion:



A metric is either elementary or composite. An elementary metric is either a raw metric, such as "number of features", or a derived metric obtained from slicing a raw metric by selection criteria. Composite metrics are obtained from existing metrics (elementary, or previously defined composite metrics) by applying some mathematical formula. The three kinds of composite metrics shown on the figure will be explained shortly.

3.6 Scopes and scope types

Every metric has a scope:

Scope of a metric

The **scope** of a metric is defined as follows:

- For a raw product metric, the type of product over which the metric is counted, such as: a feature, a class, a cluster, a system, a set of systems that have been subjected to the metric in a company, global set of all systems that have been subjected to the metric.
- For a raw process metric, the type of process on which the metric is measured, such as analysis, documentation, entire project etc.
- For a derived metric (recursively), the scope of the raw metric from which it is derived.
- For a composite metric (recursively), the largest of the scopes of all its constituent metrics, where cluster is "larger than" class and so on.

This notion also applies to measures:

> **Scope of a measure**
>
> The **scope** of a measure is defined as follows:
>
> - For an elementary measure (the application of an elementary metric, raw or derived), the set of products or processes to which the associated metric has been applied to yield the measure.
> - For a composite measure, the union of (recursively) the scopes of its constituent measures.

For both metrics and measures, the notion of scope will help us compare our quantitative assessments to some already on record. For example you may compare the value of a certain metric, such as number of invariant clauses in each class, with the values that have been archived for your project, for a reference project such as the EiffelBase library, for the previous projects of your company, or for a global set of previous projects maintained at some central location. You may also, with appropriate permissions, update such a shared archive with the values from your own measurements.

In the environment, the notion of scope will be handled in a simple way. There will be a set of predefined scopes, corresponding to software units: feature, class, cluster, system. Any scope beyond the system level will be defined by "measurement archives", as explained next.

For product scopes, we will also rely on the notion of scope type:

> **Scope type**
>
> Every product scope has a **scope type**, denoting the type of software unit of which it is an instance, for example *Class* for a class.

Class is a scope type, not a scope; a particular class defines a *scope*, of scope type *Class*. A later section introduces a set of predefined scope types: *Feature*, *Class*, *Cluster*, *System* and *Archive*.

→ "Scope types", 5.3, page 16.

3.7 Measurement archives

If you want to perform measurements on a system that you are building, the scopes of interest are of the first four types listed: *Feature*, *Class*, *Cluster* and *System*. Information on this scope will be provided by the development environment (as with ISE Eiffel 5); alternatively, you could get it simply by parsing the source of your system. But what if you also need quantitative data on other systems, if only for purposes of seeing how your results compare to the quantitative properties of other people's work?

It would be impractical in this case to require tools that have access to as much information on external systems as on your own. All we really need is a record of previous measurements on these systems. This explains the fifth scope type, *Archive*: beyond the scope of the current system, all we require to define a scope is a **measurement archive**, or just "archive" for short. This is simply a file (or part of a file) that retains, in a suitable format (XML-based), the results of measurements made earlier on one or more systems. The file can be local or accessible as a URL on the Internet.

The ability to use a measurement archive as a scope means that:

- A project may set up a measurement archive as the record of its measures.
- A department or company may set up a measurement archive for all projects on which it keeps metric information.
- The provider of the development environment, such as ISE, may publish a set of measurement archives giving metric information for reference projects, such as the EiffelBase library (designed in part as a showcase of Eiffel technology). ISE indeed intends to include a **metrics** directory, with a set of measurement archives for reference tools and libraries, in forthcoming releases of ISE Eiffel and at http://eiffel.com.

We hope that once the facilities described here are available users will adopt the practice of publishing measurement archives; it's a way of providing quantitative reference information, useful to everyone, without revealing anything critical about the actual contents of the software. (For further protection one may envision an independent group that anonymizes the data after verifying the authenticity of the source.)

4 DEFINING A COMPOSITE METRIC

You may introduce a composite metric by applying a mathematical formula to previously available metrics.

Our metrics framework includes three mechanisms for defining composite metrics: **linear combination** of existing metrics; **ratio** of two existing metrics; and **scope comparison**, the ratio of an existing metric applied to two different scopes. This is, on purpose, a restricted set of operations; we have limited ourselves, in light of the criteria listed at the beginning of this discussion, to operations that clearly make sense. We have refrained in particular from supporting arbitrary arithmetic operations, since it is not clear (for example) that multiplication of metrics is ever meaningfully. It will be easy to extend the framework to new operations supported by a valid metric theory.

← *"Desirable features of software measurements", page 2*.

Each of the three operations is defined below with an applicability precondition, a specification of its semantics, and a specification of the resulting unit.

4.1 Linear combination (weighted sum)

Notation: $\sum k_i \cdot m_i$ (where the k_i are real values)

Applicability: all the k_i are metrics with the same unit u other than *RATIO*

Semantics: weighted sum

Resulting unit: u

4.2 Ratio

Notation: a / b

Applicability: a and b are metrics, not necessarily with the same unit, but neither of them *RATIO*.

Semantics: division

Resulting unit: *RATIO*

Note that the presence of both linear combinations and ratios make it possible to define any metric of the form

$$\frac{\sum k_i \cdot m_i}{\sum l_j \cdot n_j}$$

in terms of elementary metrics m_i, n_j. If necessary, the interface can provide a shortcut to define such a formula in one step, without first defining the numerator and denominator as linear combinations.

← "REQUIREMENTS", 2, page 2.

4.3 Scope comparison

Notation: *a1 // a2*

Applicability: *a1* and *a2* are the same metric except for a possibly different scope.

Semantics: division

Resulting unit: *RATIO*

Note that since a scope may be a measurement archive, the presence of scope comparisons in our battery of metrics makes it possible to compare current measures to those of a reference archive, for example the previous measurements of a company. The result of a scope comparison doesn't have to be expressed as a raw number or percentage; other possibilities include a percentage variation, such as *–25%* to mean that the value for *a1* is 25% less than the value for *a2*.

5 METRIC CAPABILITIES

The environment must provide the following product metric facilities, all available in ISE Eiffel 5.

5.1 Units

The following predefined units should be available:

Unit	Quantities measured	Kind
CLASS	Numbers of classes or types	Product
CLUSTER	Numbers of clusters (groups of classes)	Product
COMPILATION	Numbers of compilations	Process
CONTRACT_CLAUSE	Numbers of assertion clauses	Product
LOCAL	Numbers of entities local to a routine (local variables, formal arguments)	Product
FEATURE	Numbers of features	Product
LINE	Numbers of source lines	Product
RATIO	Results of divisions	Product
SYSTEM	Numbers of systems	Product

The units listed here are **minimal** in the sense that it is not possible to add the properties measured by any two of them; for example it makes no sense to add a number of features to a number of classes.

5.2 Predefined raw metrics and selection criteria

The environment should make it possible, for any project, to apply the elementary metrics in the following table, each with an associated unit and a one-identifier name. Each of the table's major divisions starts with a raw metric, for example *Classes*, and, when appropriate, continues with selection criteria that yield derived metrics based on that raw metric, for example *Deferred_classes*.

The last column specifies the "basic scope types" of the metric; this notion, discussed in more detail in a later section, denotes the set of scope types on which the metric will need to get its information directly from the software product (rather than being simply computed by adding the corresponding values on a scope's constituents).

→ "BASIC SCOPE TYPES OF A METRIC", 7, page 24.

Basic count	Criterion	What to count	Unit	Basic scope types
Classes		Classes of a cluster or system	*CLASS*	*Cluster*
	Deferred	Deferred classes (not completely implemented, as opposed to "effective", completely implemented).		
	Invariant_ equipped	Classes having an invariant		
	Obsolete	Classes marked as superseded by newer alternatives		
Clusters		Clusters of a system, or subclusters of a cluster	*CLUSTER*	*Cluster*
Compilations		Compilations since start of project	*COMPILATION*	*System*
Comment_lines		Comment lines	*LINE*	*Feature, Class*
Comments		Comments (a comment may extend over successive lines)	*COMMENT*	*Feature, Class*
	Header_ comments	Header comments of features		
Dependents		Classes on which a class depends, or which depend on it, directly or indirectly	*CLASS*	*Class*
	Clients	Direct clients		
	Heirs	Direct heirs in inheritance structure		
	Indirect_ ancestors	Indirect parents in inheritance structure		
	Indirect_ clients	Indirect clients (clients of a direct or, recursively, indirect client)		
	Indirect_ descendants	Indirect heirs in inheritance structure		
	Indirect_ suppliers	Indirect suppliers (suppliers of a direct or, recursively, indirect supplier)		
	Parents	Direct parents in inheritance structure		
	Self	The class itself (value always 1)		
	Suppliers	Direct suppliers		

Basic count	Criterion	What to count	Unit	Basic scope types
Features		Features of a class	*FEATURE*	*Class*
	Attributes	Attributes (features represented by fields in instances of the class, as opposed to *routines*, represented by algorithms)		
	Deferred	Deferred routines (not completely implemented, as opposed to *effective*, completely implemented)		
	Exported	Features available to all clients		
	Functions	Value-returning routines		
	Inherited	Features obtained from a parent (possibly in a different form)		
	Postcondition_equipped	Routines having a postcondition		
	Precondition_equipped	Routines having a precondition		
	Queries	Value-returning features, including both attributes and functions (as opposed to *commands*, the same thing as procedures, returning a result)		
	Redeclared	Features inherited under a different specification or implementation		
	Renamed	Features inherited under a different name		
Feature_assertions		Clauses in routine's assertions	*CONTRACT_CLAUSE*	*Feature*
	Postcondition_clauses	Clauses in postcondition		
	Precondition_clauses	Clauses in preconditions		
Formal_generics		Formal generic parameters of a class	*CLASS*	*Class*
	Constrained	Formal parameters constrained by a type other than *ANY*.		
Formals		Formal arguments of a routine	*LOCAL*	*Feature*
Invariant_clauses		Clauses in invariant	*CONTRACT_CLAUSE*	*Class*
Lines		Source lines (excluding blank ones)	*LINE*	*Feature*, *Class*
Locals		Local entities (excluding *Result*)	*LOCAL*	*Feature*
Systems		Systems	*SYSTEM*	*Archive*

A few comments on specific entries:

- The list of criteria for *Features* does not include *Routines* because a routine is a feature that is not an attribute; to obtain the number of routines, just count features that do not satisfy the *Attributes* criterion.

- The selection criteria *Attributes* and *Deferred* for *Features* are, a noted earlier, not independent, since attributes may not be deferred.

- Another case of dependency: a *procedure* is never part of *Queries* but always a "command". Queries, however, include both *Attributes* and *Functions*.

- Two more cases of dependency: *Redeclared* and *Renamed* can only be satisfied for features that are *Inherited*. A feature that is not inherited, but introduced fresh in the enclosing class, is called *immediate*.

- The *Redeclared* attribute has three values: a feature is *redefined* it was effective in the parent, or it was deferred in the parent and the new class keeps it deferred with a different signature or contract; it is *effected* if it was deferred and the class makes it effective; or it may be neither of these.

All metrics listed are *product* metrics with one exception: *Compilations*, the only *process* metric, counting the number of compilations of the project (in ISE Eiffel 5, this is based on ISE's Melting Ice technology). There is room for more process metrics, such as cost estimates; this requires standard formats letting project managers enter the appropriate information, a point that future versions of the metrics policy described here may develop further.

There is also room for elementary product metrics other than those in the preceding table. In fact, every syntactic construct is a candidate for an elementary metric that simply counts the number of its occurrences; but we should limit ourselves to those that we judge interesting. The metrics literature also suggests elementary metrics assessing complexity of the control structure through properties of the control graph (McCabe), or routine coherence through such properties as the number of attributes accessed by a routine; we should only add them if we can find convincing arguments for their theoretical soundness.

5.3 Scope types

It should be possible to apply any available metric to scopes of any of the following categories:

Scope
Feature
Class
Cluster
System
Archive

Unlike the other tables, which list their entries alphabetically, the above list shows scopes types from *smaller* to *larger*. This list indeed defines an **order relation** between scope types, which follows from the containment relation of the corresponding software units: every class contains features; every cluster contains classes; and so on.

5.4 Defining, saving and loading metrics

Users should have the ability to define a new derived or composite metric by applying some of the selection criteria or composition operators to one or more existing metrics.

← "Raw metrics and selection criteria", page 6; "DEFINING A COMPOSITE METRIC", 4, page

They should also have the ability to save all current metric definitions to a file, producing a **metric definition record**. This should not be confused with a measurement archive, which records the results of *measures* resulting from applying certain metrics to certain projects.

← "Measurement archives", 3.7, page 9.

Although the concepts are distinct, ISE Eiffel lets you store a measurement archive and the corresponding metric definition record into a single file, so that when you access the archived measures you have unambiguous definition of the metrics that served to produce them.

The environment must also provide a way to add to the current metric definitions a set of metrics read from a metric definition record, available either from a local file or through a URL. As a result, users will be able to benefit from each other's metric definitions and improve them over time.

5.5 Computing, saving and aggregating measures

The environment should let users apply any available metric to any available and applicable scope.

← "Scopes and scope types", 3.6, page 8.

It should also let them save the resulting measure to a measurement archive.

These facilities will also enable the creation and use of new **scopes**. In particular:

- Giving a user read permissions on a measurement archive allows him to access the associated scope, for example to compare his current project's value to those on record. This will be called **archive access**.

- Giving the user write permissions makes it possible to update the measurement archive with new measures: **archive update**.

- It should also be possible to combine several measurement archives into one. This operation is **archive aggregation**; you can use it for example to create cumulative records of measures at any level: a team; a department; a company; or even a group that maintains a global record of all submitted measures.

Conceptually, a measurement archive consists of quadruples of the form

[*system*, *metric*, *scope*, *value*]

where the *system* uniquely identifies the system to which the measures apply. The *system* may for example consist of a system name with a version number. No two entries of a given archive may have the same [*system*, *metric*, *scope*] part. To maintain this invariant:

- If an entry with the same [*system*, *metric*, *scope*] value is already present, the update operation must replace it, not add a new entry.

- The aggregation operation, when merging two archives, must remove any conflicts, for example by renaming the *system* part.

It is the responsibility of the archive maintainer to decide, through the appropriate choice of *system*, which measurements should add entries and which replace existing entries. This is important because some metric computations — for example to compute averages, e.g. the average number of features per class — will add all the *value* items, for a certain *metric* and certain scope types, over every applicable *system*. The wrong choice of *system* could skew the results by counting a system several times, or counting different systems as one. Assume for example you perform measurements on a new release. If you store the measures into an existing archive, which holds measures performed on the previous versions, you should decide what you prefer:

→ *This additive property will follow from clause* *3 of "How to compute a measure", page 24*.

- Using a new *system* means that both the new and old measures will be kept, and both will affect any cumulative analysis such as averages and statistics. This means that you consider the upgrade as defining a new product, whose properties will be counted in addition to those of the old one. For a mere incremental upgrade to your system, this policy would be inappropriate, as it would distort cumulative measures involving several systems by counting yours several times.
- Keeping the same *system* means that the new measures will override any previous measures, and so that your system's properties will be counted only once. This is appropriate for a routine revision, but not for a redesign if you want to retain the original measures in the record.

The *metric* identification must also be handled with care since different measures may, especially if they come from different projects, refer to different derived or composite metrics that accidentally use the same name.

To address this problem, ISE Eiffel 5 always stores the current metric definition record (the definition of the current metrics) as part of any measurement archive. This way, a metric name is never ambiguous: it either refers to a raw metric (which has a universal name), or to a derived or composite metric uniquely identified in the metric definition record.

← *"Predefined raw metrics and selection criteria", page 12*

The operation of measurement archive aggregation as defined above (merging two measurement archives) must apply a simple renaming algorithm to remove name clashes if the archives being aggregated use the same metric name for two different metrics.

6 USER INTERFACE BASICS

6.1 Overview

In ISE Eiffel 5, the user interface permitting all the operations described in the preceding sections is part of the basic development environment. It is provided by a tab "**Metrics**", appearing along with others such as "**Class info**" (formats that yield various views of classes, such as interface specification and inheritance structure) and "**Diagram**" (graphical system representation, reverse-engineered or serving to generate the software through forward engineering).

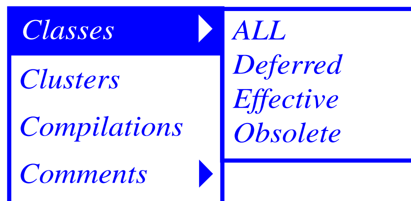
6.2 Predefined metrics

The environment should by default provide a set of elementary metrics. They must include all the raw metrics defined in the earlier table (*Classes*, *Compilations*, *Features* etc.).

← *"Predefined raw metrics and selection criteria", page 12*

The list of predefined metrics should also include some derived metrics resulting from applying some of the criteria of the table to the corresponding raw metrics to cover some commonly used notions, for example the number of commands, queries, routines and attributes in a class. As the resulting number of predefined metrics is large the

environment must, wherever it allows selection of a metric through a menu, list only the raw metrics in a primary menu, and use submenus to show their derived metrics, as in:



6.3 Defining a new metric

The next sections describe the process for defining new metrics. The user interface sketches will be replaced, in the final version of this note, by screenshots of the actual tools.

You may define a new derived metric, or a new composite metric. In both cases you will fill in two text fields with, respectively:

← *If you forgot the difference, see the figure on page 8.*

- A short name, normally a single identifier.
- (Optionally) a long name, or "comment".

You will never need to provide a unit; instead the tool will deduce the unit from the metric's definition, and display it.

6.4 Defining a new elementary (derived) metric

To define a new elementary metric, derived from one of the raw metrics, you specify that metric, and the selection criteria that you want to apply:

New derived metric Short name:

Comment:

Raw metric: *Class* Unit: *CLASS*

Count if: ◉ At least one ○ All of the following are met

○ Deferred	○ Effective	◉ Ignore
○ Obsolete	○ No	◉ Ignore
○ Invariant equipped	○ No	◉ Ignore

OK

After the name and the optional longer name ("Comment"), you will select a raw metric; the unit immediately appears, as well as the set of criteria with their possible values, from which you choose the desired "and" or "or" combination. Criteria marked "Ignore" do not take part in the selection.

6.5 Defining a composite metric

To introduce a new composite metric, you will provide the mathematical formula that defines it in terms of existing metrics. The dialog that enables you to enter this definition lets you choose between the three composition operations described in an earlier section:

← "DEFINING A COMPOSITE METRIC", 4, page 10.

- **New linear metric**: new metric of the form $\sum k_i \cdot m_i$ for existing non-ratio metrics m_i.
- **New ratio metric**: new metric of the form a / b for existing non-ratio metrics a and b.
- **New comparison metric**: new metric comparing the values of an existing non-ratio metric

These conventions imply that you can only define a metric of the general form stated earlier:

$$\frac{\sum k_i \cdot m_i}{\sum l_j \cdot n_j}$$

by defining two separate linear metrics for the numerator and denominator, then defining a ratio metric for the desired result.

We will now review the user interface setups for these three cases.

6.6 Defining a new linear metric

The dialog that lets you define a new linear metric of the form $\sum k_i \cdot m_i$ will present you with a dialog of the following form:

New linear metric Short name:

Comment:

1 * *Attributes* ▼ **Add** **Remove last** **OK**

Metric definition: Unit:

OK

You will use the line after the short name and comment fields to enter the successive terms of the linear metric (the successive k_i and m_i in $\sum k_i \cdot m_i$).

The first element on that line is a field for entering a k_i multiplier; it's initialized to 1, the default and most common value.

The second element lets you choose a metric m_i by presenting you with a menu listing the available metrics, starting with *Attributes* on the figure. Initially it includes all the metrics defined so far, elementary and composite, whose unit is anything other than *RATIO*. After you have chosen the first term, the Unit field will display the name of the term's unit, and the menu will only show metrics that use the same unit.

Pressing the **Add** button will add the current term (multiplier times the chosen metric) to the current definition. That definition will be updated on the last line (after "Metric definition:") so that, after adding two metrics, *Program_lines* with coefficient 1 and *Comment_lines* with coefficient —1 you would get the following:

New linear metric Short name: *Non_comment_lines*

Comment: *Number of source lines, excluding all comments (lines starting with - -)*

1 * *Attributes* ▼ **Add** **Remove last** **OK**

Metric definition: *Program_lines – Comment_lines* Unit: *FEATURE*

OK

with the metric definition simplified by the tool from *1 * Program_lines + (–1) * Comment_lines*.

The **Remove last** button removes the last entered term. (The only way to remove a term other than the last is to click **Remove last** as many times as needed, and reënter the subsequent terms).

You never need to specify a unit: as soon as you have chosen the first term (either the first time around, or if you do it again after using **Remove last** to cancel all terms previously entered), the list of menu choices in the second control will be limited to the metrics having the same unit. The tool displays that unit in the unit field (bottom right on the figure).

Clicking the **OK** button adds the metric to those already defined.

6.7 Defining a new ratio metric

The dialog that lets you define a new ratio metric presents you with two menus: one to choose the numerator metric and one to choose the denominator metric. Each includes the list of all available non-*RATIO* metrics. (In the absence of a calculus of units, we do not allow computing the ratio of two ratios.)

New ratio metric Short name:

Comment:

Numerator: *Attributes* Denominator: *Attributes*

Metric definition: Unit: *RATIO*

Display as percentage

OK

The tool will display *RATIO* in the unit field.

6.8 Defining a new comparison metric

The dialog that lets you define a new comparison metric asks you for three pieces of information:

- The base metric, to be selected from a menu of all non-*RATIO* metrics.
- The scope of the numerator (see below about how to select a scope).
- The scope of the denominator.

New comparison metric Short name:

Comment:

Base metric: *Attributes*

Metric definition: Unit: *RATIO*

Numerator scope: *Feature* Denominator scope: *Feature*

Display as percentage

OK

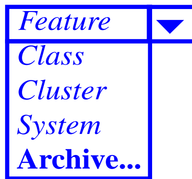
6.9 Loading and saving metrics

The user interface provides tools for the operations, previously noted, of adding the current metric definitions to a metric definition record and loading a set of metric definitions from a metric definition record, found either in a file or on the Internet at a certain URL.

6.10 Selecting a scope

To select a scope, you will get a menu listing the entries corresponding to the predefined scopes (*Feature* etc., see earlier list), including **Archive...** as the last possibility:

← "Scope types", 5.3, page 16.



Choosing **Archive...** lets you browse to find a measurement archive file, either local or, if available on the Internet, given by its URL. Any resulting measure will be drawn from the information recorded in that file, rather than from the current system.

← "Measurement archives", 3.7, page 9.

6.11 Applying metrics and saving measures

To perform a measure, you will simply choose a metric and a scope, and request the computation.

You may save the result into a measurement archive as explained earlier. A tool will also be available for archive aggregation.

6.12 Specifying a reference scope

In the course of computing measures by applying metrics to your project, you may find it useful to compare the values obtained with those of a reference scope — another project, or a measurement archive covering earlier projects.

You can obtain this effect by defining comparison metrics, as discussed. This is not convenient, however, if you want to apply the comparison to *all* your measurements, or if you want to change the scope used as a reference.

To support these goals, the environment provides an option that lets you choose a **reference scope**, normally a measurement archive, found either in a file or at a given URL. If you have set the reference scope, the user interface will show, every time you compute a measure, the corresponding value for the reference scope. It may also show the relationship in the form of a percentage.

This information is for display only and will not be included if you save your current measures into a measurement archive.

7 BASIC SCOPE TYPES OF A METRIC

We need a clarification regarding the connection between metrics and scopes. This is a technical point and you can start using the tools without understanding it.

The reason for this notion, "basic scope types of a metric", is that some metrics are *additive* in a simple way across the constituents of a scope, but others are not. "Constituents" of a scope are the subscopes according to the order relation defined earlier between scopes: systems for an archive, clusters for a system, classes for a clusters, and features for a class.

← *"Scope types", 5.3, page 16*.

To see the need for defining basic scope types, take as an example the elementary metric *Lines*, denoting the number of source lines in a certain scope. Then:

- The metric is additive at the cluster level, in the sense that the number of source lines of a cluster is the sum of the corresponding numbers for its constituent classes.
- The metric is similarly additive at the level of systems (add *Lines* over constituent clusters) and archives (add *Lines* over constituent systems).
- The metric is, however, **not additive** at the class level, since the number of source lines of a class isn't just the sum of numbers of source lines of its features; we must also count the lines of Indexing clauses, Invariant clauses and other elements of a class that are not in a feature.

We will say that the scope types *Feature* and *Class* are the *basic scope types* of the metric *Lines*. This means the definition of the metric must, in specifying how to compute a measure for that metric over a particular scope, include three different algorithms, as follows.

How to compute a measure

The following algorithm determines how to obtain a measure by applying a metric to a scope:

1 • If the scope's type is one of the metric's basic scope types, *Feature* or *Class* in the example, the metric's specification must include a predefined mechanism to compute the measure. That mechanism will directly access properties of the software — through the development environment (as in ISE Eiffel), or by directly analyzing the source code.

2 • If the scope's type is **smaller** than all of the metric's basic scope types — referring to the order relation between scopes: *Feature* less than *Class*, *Class* less than *Cluster* and so on — the measure will always be zero. This cannot happen with the *Lines* metric, since the smallest of all scope types, *Feature*, is one of its basic scope types. But if requested to apply the metric *Parents* (number of parents) to a feature, we should return zero, since the *Class* is the only basic scope type for this metric and *Feature* is smaller than *Class*. The reason for this is obvious: we are asked to compute the "number of parents" for a feature, but the smallest unit for which this metric is defined is *Class*, so the only appropriate answer is zero. (This rule is important in connection with the next case.)

3 • Otherwise (scope's type not one of the metric's basic scope types, but **bigger** than the smallest one), the measure is the sum of the measures obtained, recursively, by applying the metric to all of the scope's constituents. For example to compute *Lines* — the number of lines — over a cluster is simply to compute it for each of its constituent classes and add the results. This rule is recursive: to compute *Lines* over a system, we add the results of applying the rule to each one of its constituent clusters.

To apply this definition, we need to know how to determine the basic scope types of any metric. Here is the rule:

Basic scope types of a metric

Every metric has a set of basic scope types, defined as follows:

A • The basic scope types of an elementary metric are part of the metric's definition; they appear in the table of elementary metrics.

← "Predefined raw metrics and selection criteria", 5.2, page 12.

B • The basic scope types of a composite metric include (recursively) the basic scope types of all its constituent metrics.

To understand case B, which defines the set of basic scope types of a composite metric as the union of those of its constituents, we need to refer to all clauses 2 and 3 of the preceding rules. Assume that you want to define a composite metric "Number of contracts" as

Precondition_clauses + Postcondition_clauses + Invariant_clauses

← *As defined in* "Predefined raw metrics and selection criteria", 5.2, page 12.

This is a legitimate combination since all the constituent metrics have the same unit, *CONTRACT_CLAUSE*, and indeed represent quantities (numbers of contract clauses) that it is meaningful to compare or add.

Case B of the definition of “Basic scopes of a metric” tells us that the basic scopes of the metric include both *Feature*, the basic scope type of the metrics *Precondition_clauses* and *Postcondition_clauses*, and *Class*, the basic scope type of the metric *Invariant_clauses*. This is what we want since the metric is meaningful for a feature (where it will only count precondition and postcondition clauses) and a class (where it will count all three kinds).

Assume we want to compute this quantity for a certain **feature**:

- The definition of the metric tells us that we must compute the numbers of precondition, postcondition and assertion clauses for the class, and add the results.

- For the *Precondition_clauses* and *Postcondition_clauses*, the only basic scope type is *Feature*, which is our scope of interest, so clause 1 of the rule on “How to compute a measure” tells us to that we must count the precondition and postcondition clauses for all the features of the class.

←*Again as defined in “Predefined raw metrics and selection criteria”, 5.2, page 12.*

- For the *Invariant_clauses* metric, our scope of interest, *Feature*, is smaller than the metric’s only basic type scope, *Class*, so clause 2 of the rule tells us to ignore this metric by returning a zero. Without this clause, we would be faced with a logical contradiction.

If, however, we want to apply the metric to a certain **class**:

- For the *Precondition_clauses* and *Postcondition_clauses*, since the only basic scope type of these metrics is *Feature* and our scope is bigger, clause 3 of the rule tells us to count precondition and postcondition clauses for all of the features of the class, and add the results.

- For *Invariant_clauses*, clause 1 applies: we count the number of invariant clauses for the class, and add it to the previous result.

As this example indicates, we need the rules to make the definition and application of metrics consistent. This enables the environment to provide users with a set of metrics which they can trust — even if they are not aware of the detail of the rules.