Università degli Studi di Bologna
DEIS

A Survey of Software Metrics

Fabrizio Riguzzi

July 1996

DEIS Technical Report no. DEIS-LIA-96-010 *LIA series no. 17*

A Survey of Software Metrics

Fabrizio Riguzzi

Dipartimento di Elettronica, Informatica e Sistemistica,
Università di Bologna
Viale Risorgimento, 2 (Bologna) Italy
friguzzi@deis.unibo.it

July 1996

Abstract. In this paper we present a survey on the research field of software metrics. We start by considering the reasons why measurement of software was introduced, then we describe what are the attributes of software and of the software process that are the objects of measurement. A general theory of measurement is presented that was first applied to software measurement by [Fenton 94]. The two types of metrics validation, empirical and theoretical, are discussed. Then a number of metrics for the different development phases is presented in details. We discuss function points for the requirements phase and the metrics for choesion and coupling defined in [Briand et al 94] for the design phase. For the implementation phase we describe the suite of metrics for OO designs by [Chidameber, Kemerer 94], Lines of Code [Conte et al. 86], Software Science [Halstead 75] and Cyclomatic Number [McCabe 76].

Keywords: *Software engineering, software metrics.*

1. Introduction

The field of software metrics is a relatively young one, whose origins can be found in the work by Halstead published in 1972. From then on, the interest in software metrics has steadily increased because they have been recognized as an helpful instrument for managing effectively the software process. Software metrics allow to use a real engineering approach to software development, providing the *quantitative* and objective base that software engineering was lacking. In fact, their use in industry is becoming more and more widespread.

As regards the research in software metrics, it has undergone a great evolution: in the first period the focus was very much on inventing new metrics for the different attributes of software, without so much regard for the scientific validity of the metrics. In recent times instead, a lot of work has been done on how to apply the theory of measurement to software metrics and how to ensure their validity.

The survey is organized as follows. In Section 2 we discuss in more detail the different uses of measurement. In Section 3 we concentrate on the objects of measurement, the attributes of software. In particular, the complexity attribute is considered, which influences many other attributes. In Section 4 presents the theory of measurement and section 5 how to validate a metric. Section 6 contains the description of six of the most important and diffused metrics, classified according to the phase of software development in which they can be collected. For the requirements phase we present function points, for the design phase the metrics by [Briand et al 94]. For the implementation phase we describe the suite of metrics for OO designs by [Chidameber, Kemerer 94], Lines of Code [Conte et al. 86], Software Science [Halstead 75] and Cyclomatic Number [McCabe 76]. In Section 7 we summarize the main points treated

2. Uses of measurement

Measurement has two broad uses: for *assessment* and *prediction* [Fenton 94].

Examples of using metrics for assessment are:

1. monitoring the advancement of a project in order to take the appropriate corrective decisions,
2. evaluating a software product or process.

When using measurement for prediction, the value of an attribute A is given by a mathematical model, very often stochastic rather than deterministic, that relates A to the measurement of other attributes. Predictive measures can be used for:

1. planning the resources and time needed for a certain project,

2. predicting the outcome of a project, in terms of size, quality or other attributes of the delivered software.

In both cases, software metrics can provide a *quantitative support* to decisions that were previously taken on the basis only of subjective factors. The manager can now take the decision of which tool or language to use or of the kind of process to adopt using data from previous projects instead of relying only on intuition. The researcher instead can use software metrics to demonstrate empirically that a certain methodology is the best one given certain conditions.

3. What to measure

In order to measure, we need to identify an *entity* and a *specific attribute* of it. It is very important to define in a clear way what we are measuring because otherwise we may not be able to perform the measure or the measures obtained can have different meaning to different people.

The entities considered in software measurement are [Fenton 94]:

1. *processes*: any activity related to software development,
2. *product*: any artifact produced during software development,
3. *resource*: people, hardware, or software needed for the processes.

The attributes of entities can be internal or external.

- *Internal attributes* of an entity can be measured only based on the entity and therefore the measures are direct. For example size is an internal attribute of any software document.
- *External attributes* of an entity can be measured only with respect to how the entity relates with the environment and therefore can be measured only indirectly. For example, reliability, an external attribute of a program, does not depend only on the program itself but also on the compiler, machine and user. Productivity, an external attribute of a person, clearly depends on many factors such as the kind of process and the quality of the software delivered.

In table 1 we can see examples of entities and attributes for the different phases of software development.

As can be seen from the table, measurement can take place in all the different phases of software development: Requirement Analysis, Specification, Design, Coding and Verification. Undoubtedly, measurement is most useful if carried out in the early phases. But these are the phases in which it is more difficult and in fact most measures have been defined for the Coding phase.

Entity	Internal Attributes	External Attributes
Product		
Requirements	Size, Reuse, Modularity, Redundancy, Functionality	Understandability, Stability
Specification	Size, Reuse, Modularity, Redundancy, Functionality	Understandability, Maintainability
Design	Size, Reuse, Modularity, Coupling, Cohesion	Comprehensibility, Maintainability, Quality
Code	Size, Reuse, Modularity, Coupling, Cohesion, Control Flow Complexity	Reliability, Usability, Reusability, Maintainability
Test set	Size, Coverage level	Quality
Process		
Requirements analysis	Time, Effort	Cost effectiveness
Specification	Time, Effort, Number of requirements changes	Cost effectiveness
Design	Time, Effort, Number of specification changes	Cost effectiveness
Coding	Time, Effort, Number of design changes	Cost effectiveness
Testing	Time, Effort, Number of code changes	Cost effectiveness
Resource		
Personnel	Age, Cost	Productivity, Experience
Team	Size, Communication Level, Structure	Productivity
Software	Size, Price	Usability, Reliability
Hardware	Price, Speed, Memory size	Usability, Reliability

Table 1: entities and attributes (adapted from [Morasca 95]).

The external attributes are clearly the most interesting from the point of view of the manager, but they can be measured only indirectly. For example, productivity of people can be measured as the ratio of size of product delivered (an internal code attribute) and effort (an internal process attribute). Furthermore, external attributes are difficult to define: it is rare that there is a consensus on the definitions of these attributes. For example, quality can be defined as the ratio of faults discovered during formal testing (an internal process attribute) and size, measured by KLOC (Kilo Lines Of Code) [Fenton 94]. In alternative, quality can be considered as a very high-level attribute constituted by a combination of reliability, availability, maintainability and usability. In turn, maintainability comprises understandability, modifiability and testability [Henderson-Sellers 96] (Fig. 1). Moreover each of these component is influenced by complexity. So we see that external attributes are not isolated from each other but are closely related.

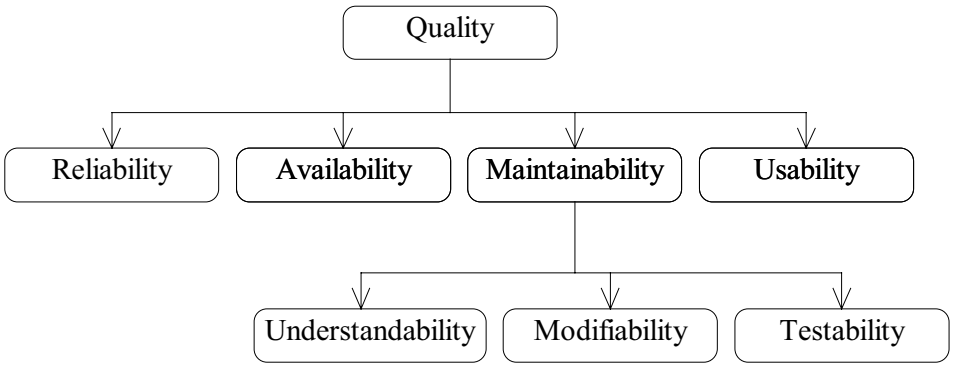

Figure 1 (from [Henderson-Sellers 96]).

The term complexity deserves a special attention. In table 1 we have cited control flow complexity as a product internal attribute and many authors call this simply complexity. For other authors instead complexity is a very high level term that comprehends nearly all aspects of software. [Fenton 91] distinguishes three kinds of complexities: computational, psychological and representational. We will concentrate only on the psychological one that is complexity as perceived by man. This kind of complexity comprehends structural complexity, programmer characteristics and problem complexity. Programmer characteristics are hard to measure objectively while little work has been done to date on measures of problem complexity. Structural complexity instead has been studied extensively because it is the only component of psychological complexity that can be assessed objectively. Many metrics have been proposed for structural complexity and they measure a number of internal attributes of software. *Structural metrics* can be divided in intramodule (or just module) metrics and intermodule metrics. *Module metrics* are focused at the individual module level (subprogram or class) and comprehend: size metrics, control flow complexity metrics, data structure metrics and cohesion metrics. *Intermodule metrics* measure the interconnections between modules of the system and are constituted by coupling metrics. Fig. 2 represents graphically this classification of complexity.

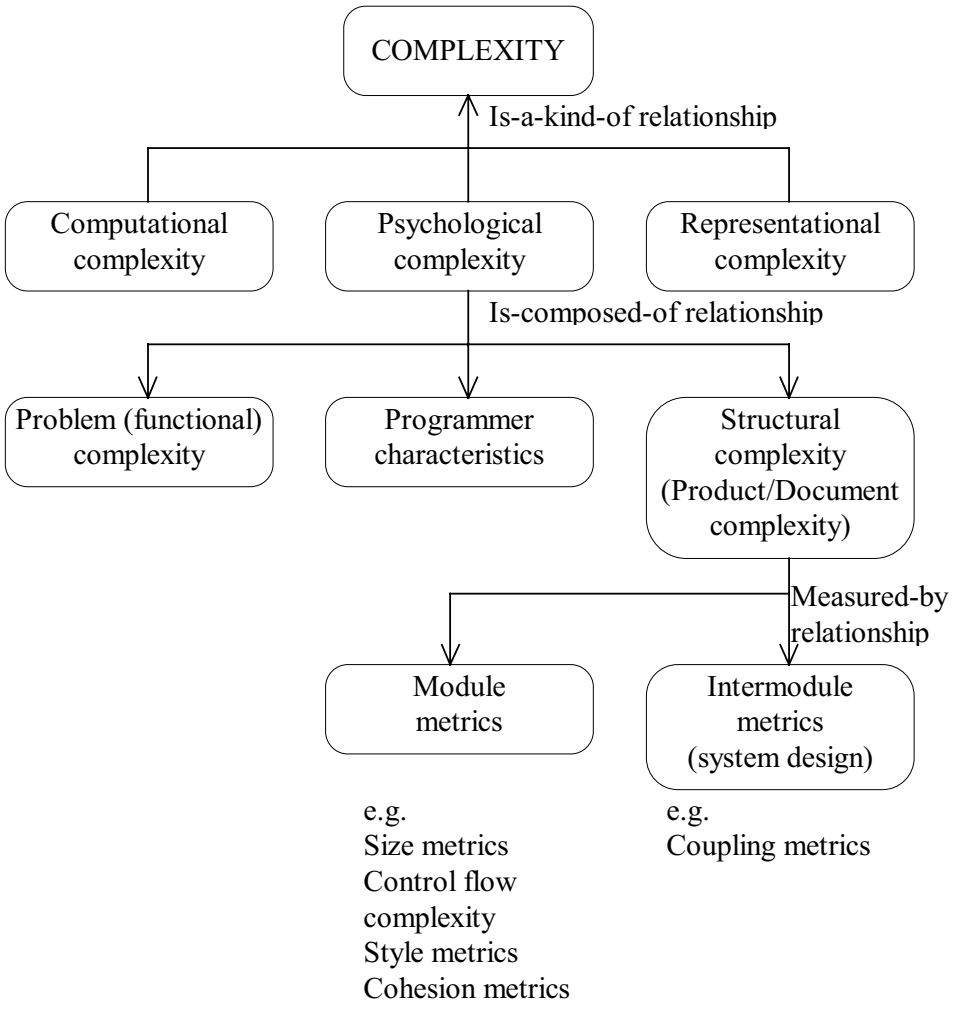

Figure 2 (from [Henderson-Sellers 96]).

[Weyuker 88] has proposed a set of axioms that any metrics for structural complexity should satisfy. The purpose of these axioms is to provide an instrument for the evaluation and the comparison of the new metrics that are proposed.

Let P and Q be program bodies and $|P|$ and $|Q|$ their complexities.

Property 1: *A measure must not be too coarse (1).*

$\exists P, Q \quad |P| \neq |Q|$

because a measure which rates all programs as equally complex is not really a measure.

Property 2: *A measure must not be too coarse (2).*
Given a non-negative number c, there are only finitely many programs of complexity c
In this way we ensure that there is sufficient resolution in the measurement scale to be useful, for example a metrics with only 2 values is not likely to be useful.

Property 3: *A measure must not be too fine.*
There are distinct programs *P* and *Q* such that $|P| = |Q|$.
For example, if we consider as a metric the binary representation of the program, it would fail to satisfy this property.

The first three properties are properties that every measures should satisfy and do not express nothing particular about measures of complexity.

Property 4: *Functionality is not the same as complexity.*
$\exists P, Q$ such that they are functionally equivalent but $|P| \neq |Q|$
There is a difference between functionality and complexity: even if two programs compute the same function, their complexity depends on the implementation.

Property 5: *Monotonicity with respect to program concatenation (; operator).*
$\forall P, Q \quad |P| < |P; Q| \; and \; |Q| < |P; Q|$

Property 6: *The contribution of a program body to overall complexity may depend on the rest of the program bodies.*
a) $\exists P, Q, R \quad |P| = |Q| \; and \; |P; R| \neq |Q; R|$
b) $\exists P, Q, R \quad |P| = |Q| \; and \; |R; P| \neq |R; Q|$
This property makes the measure not additive:
$|P; Q| = |P| + |Q|$
is not always true.

Property 7: *Sensitiveness to the order of statement.*

$\exists P, Q$ such that Q is formed by permuting the order of statements of P and $|P| \neq |Q|$.

Property 8: *Renaming.*

If P is a renaming of Q than $|P| = |Q|$.

The names of identifiers in a program can have a strong impact on understandability and this can be considered a kind of complexity. However Weyuker does not take this factor into account because it is very difficult to quantify it in an objective way.

Property 9: *Interaction increases complexity.*

$\exists P, Q \quad |P| + |Q| < |P; Q|$

The complexity of a program formed by concatenating two programs can be, in some cases, greater then the sum of their complexities because, when concatenated, there can be additional interactions between them.

Many authors have criticized these properties (we will see in the next section the critique by Fenton) but their importance lies in the fact that they have provided a starting point for the discussion about how to validate theoretically a metric and they have used by many authors to assess the metrics they were proposing.

4. Theory of measurement

[Fenton 94] has underlined the necessity to find more firmly software metrics on measurement theory. This has not been done in most of the previous work in the field, with the result that many metrics are theoretically flawed. Therefore in the following we are going to provide a summary of the key concept of measurement theory.

Measurement is the process of assigning numbers or symbols to attributes of entities in the real world in such a way as to describe them according to clearly defined rules. This assignment must preserve any intuitive and empirical observations about the attributes and entities. For example, when measuring the height of humans, bigger numbers have to be assigned to taller people. Moreover we need a clear model of the attribute in order to avoid ambiguities: in the example of the height of humans, the model might specify a particular posture and whether or not to include hair height or allow shoes to be worn. By specifying a model, we define as well the relations that hold for humans with respect to height (these are the empirical relations).

In the following we describe the representational theory of measurement that contains the rules that a measure has to satisfy in order to correctly represent an attribute.

Empirical Relations System: is a couple (C,R) where C is the set of entities and R is the set of empirical relations for the attribute considered. This system has the aim of defining the intuitive meaning of an attribute by identifying the empirical relations that hold for that attribute. For example, for the height of humans the empirical relations are "is taller than", "is tall" and "much taller than".

Representation Condition: measuring an attribute characterized by an empirical relation system (C,R) means associating (C,R) with a numerical relation system (N,P) by using a mapping M. In particular, M maps entities of C to numbers of N and relations of R to relations of P so that every empirical relation is preserved. This is the so called representation condition and M is called a representation. If M maps the empirical binary relation $\prec$ into the numerical relation $<$, then the representation condition can be expressed more formally by

$$x \prec y \Leftrightarrow M(x) < M(y)$$

The representational approach to measurement helps to avoid the temptation to define a poorly understood, but intuitively recognizable, attribute in terms of some numerical assignment. This is one of the most common failings in the software metrics work: for example McCabe's complexity [Mc Cabe76] and Halstead's E [Halstead 75].

Scale types and Meaningfulness: each representation is characterized by a scale type. The existing scale types are listed in Table 2 , together with some examples and the admissible transformation.

Scale Type	Examples	Admissible Transformation
Nominal	Labeling/Classification	One-to-one mappings
Ordinal	Preference, Rankings	Monotonic Increasing Transformations
Interval	Calendar time, Centigrade/ Fahrenheit temperature	M'=aM+b (a>0)
Ratio	Length, Weight, Time intervals, Absolute temperature	M'=aM (a>0)
Absolute	Counting entities	M'=M

Table 2 (from [Morasca 95]).

The admissible transformation is the factor that distinguish the different scale types: by means of this transformation we can change from a valid representation of an empirical relation system to another. In fact, given an empirical relation system, there can be different representations but they are all related by an admissible transformation. For example, we can measure the height of people in inches or meters, and in this case the relation is M'=cM with c=2.54.
This rigorous definition of the scale type allows us to determine what statements about measurement are *meaningful*: a statement is meaningful if its truth or falsity remains unchanged under any admissible transformation of the measure involved. For example, it is not meaningful to say that "failure x is twice as critical as failure y" if we have an empirical relation system for failure criticality that admits only an ordinal scale. In fact, we could have two different valid ordinal measures M and M' for which M(x)=6,M(y)=3, and M'(x)=10,M'(y)=9. In this case the statement is true for M but not for M'.
The notions of meaningfulness allows us as well to determine what statistical operations can be done on measures. For example, in order to calculate the average of a set of data, it is meaningful to use means for data measured on a ratio scale but not on an ordinal scale, while medians are meaningful for ordinal scale but not for a nominal one.
For a certain empirical relation system different scale types can be used: the conditions under which a certain scale is possible for a relation system are given by the Representation Theorems. For example, Cantor's theorem gives condition for real-valued ordinal-scale measurement when we have a countable set of entities C and a binary relation b on C:
Cantor's Theorem: The empirical relation system (C,b) has a representation in ($\Re$,<) if and only if b is a strict weak order. The scale type is ordinal when such a representation exists. The relation b being a "strict weak order" means that it is:

1 *asymmetric* (xRy implies that it is not the case yRx) and
2 *negatively transitive* (xRy implies that for every z belonging to C, either xRz or zRy).

In the next subsections, we will see how the theory of measurement can be applied to the study of metrics for software complexity and to the analysis of Weyuker's properties.

4.1 Measurement theory applied to complexity measures

Complexity, as we have seen, is probably the most important attribute of software because it influences a number of other attributes such as maintainability, understandability, modifiability, testability and, ultimately, cost. As such, it has been extensively studied in the literature and for many years researchers have

sought to characterize complexity with a single number. [Fenton 94] has demonstrated that this is not possible. He has also shown that the axiomatic approach for complexity given by Weyuker has serious weaknesses because it attempts to characterize incompatible views of complexity.

4.1.1 A single measure for complexity does not exist

In order to prove that a single real valued metric for complexity cannot exist, Fenton demonstrates that it does not exist even for a particular kind of complexity, control flow complexity. Nearly all the metrics defined for this attribute are based on the following two hypothesis:

Hypothesis 1: the empirical relation system relative to control flow complexity contains an empirical relation b "less complex than": $(x,y) \in b$ if there is a consensus that x is less complex than y.

Hypothesis 2: the measure proposed $M: C \rightarrow \Re$, where C is the class of programs, is a representation of complexity in which b is mapped into <.

The problem is that C is not totally ordered with respect to b, while $\Re$ is totally ordered with respect to <. This means that for some couples of programs, it is not possible to establish which is the most complex, while given two real numbers one is always smaller than the other.

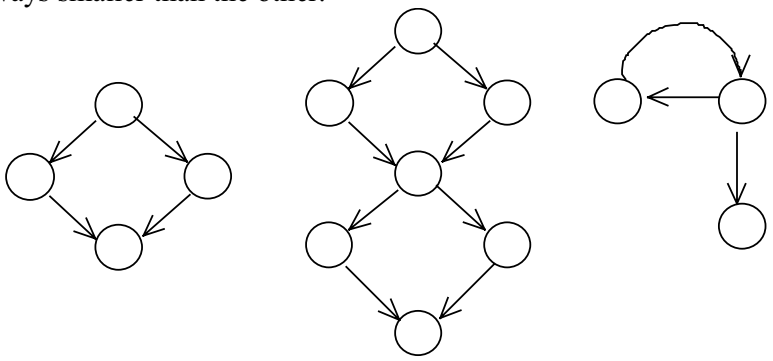

Figure 3 (from [Fenton 94])

For example, in figure 3, it seems plausible that $(x,y) \in b$, while x and z and y and z appear to be incomparable: there is no order that would be accepted by everybody. Therefore $(x,z) \notin b$, $(z,x) \notin b$,$(y,z) \notin b$ and $(z,y) \notin b$. But if we measure complexity using real numbers and the relation <, it will be either M(x)<M(z) or M(z)<M(x) and, according to the representation condition, $(z,x) \in b$ or $(x,z) \in b$. Therefore the measurement mapping forces an order that is not present in C.

While hypothesis 1 is plausible because it does not state anything about b being totally ordered, hypothesis 2 should be dismissed because it violates the representation condition.
This can be stated formally as:
Theorem 1: Assuming hypothesis 1, there is no general notion of control-flow complexity that can be measured on an ordinal scale in $(\Re, <)$.
The demonstration uses Cantor's theorem and consists in proving that b is not a strict weak order. In fact there exist a z such that $(x,y) \in b$ but $(x,z) \notin b$ and $(z,y) \notin b$ and this violates condition 2 of the definition of strict weak order.

4.1.2 Weyuker's properties

The weakness of Weyuker properties lies in the fact that they try to characterize all the different aspects of complexity with a single metric. Let us consider properties 5 and 6: property 5 asserts that adding code to a program cannot decrease its complexity. This corresponds to considering the *size* of the program as an important factor characterizing complexity. This property however does not take into account understandability of the program because, in some cases, if we see a bigger part of a program we can understand it better. Thus property 5 is satisfied by a "size" type complexity measure but not by a "understandability" type complexity measure.
Property 6 instead asserts that there are programs of equal complexity that when separately concatenated to a third one give programs of different complexity. This is due to the fact that different interactions can arise in the two cases that can be differently hard to understand. Therefore this property has to do with understandability rather than size.
Thus, these two properties are related to incompatible views of complexity and cannot be satisfied by a single measure. Even if this is not a formal argument, [Zuse 90] has proved that Weyuker's axioms are contradictory with respect to the representational theory of measurement because, while property 5 requires a ratio scale for the measure, property 6 explicitly excludes the ratio scale.

5. Metrics validation

At the moment there is not a universally accepted notion of validity. A number of approaches for the validation of metrics have been proposed but the discussion is still open.
In much of the software engineering literature validation has been performed either analytically, by using Weyuker's properties, or empirically, by showing that

the metric correlates with some other existing measure. However, these forms of validation are not completely satisfactory. We have seen that Weyuker's properties are not consistent with respect to measurement theory. Showing that a new measure correlates with an existing one just means that the new measure is the independent variable in a prediction system for the old measure. The experimental work would be scientifically correct if the prediction system, the experimental hypothesis and the dependent variable were specified in advance. In most of the works this is not done and they consist in a search for fortuitous correlations with any data that is available.

In recent times, many authors have tried to overcome these problems by proposing new validation frameworks.

According to [Fenton 94], validating an assessment type measure requires to demonstrate empirically that the representation condition is satisfied for the attribute being measured. For a predictive measure, you have to define clearly the hypothesis that has to be proved and all the components of the prediction system before designing the experimental plan.

[Kitchenam et al. 95] propose a structural model of software measurement which describes the elements of measurement (entities, attributes, units, scale types, values, properties of values, instrument) and their relationship with one another. When a new measure is created, it is necessary to give definition models for the unit, the instrumentation, the attribute relationship (if the measure is given by a functional relationship involving other attributes), the measurement protocol and the entity population.

Using this structural model, they distinguish two kinds of validation:

- *theoretical validation*, which consists in verifying that the measure does not violate any necessary properties of the elements of measurements or the definition models (e.g. a compound unit must be constructed by transformation permitted by the scale type of its components)
- *empirical validation*, which provide corroborating evidence of validity by showing that measured values of an attribute are consistent with values predicted by models involving the attribute (e.g., the measured distance from an origin to a point is the square root of the sum of the x and y coordinate values).

Real validation is only theoretical, empirical methods can only corroborate the validity of a measure.

[Briand et al. 96] propose a mathematical framework for the definition of software attributes like complexity, coupling, cohesion, length and size. For each attribute they give a number of properties that a measure for that attribute should satisfy. Therefore they follow the axiomatic approach initiated by Weyuker, but they use a

mathematical model of systems and modules, which allows them to state properties in a formal and rigorous way.

6. Examples of metrics

We will present in this section six metrics chosen among the most widely used and known ones. We will group these metrics according to the phase of software development in which they can be applied.
In the *requirement phase*, function points can be used to estimate the size of the resulting system.
For the *design phase*, we will present the metrics for cohesion and coupling defined by [Briand et al. 94]. More than 500 metrics have been proposed for the *implementation phase*. We will see: the suite of metrics for OO designs by [Chidameber, Kemerer 94], Lines of Code, Halstead's Software Science and McCabe's Cyclomatic Number.

6.1 Function Points

Function Points [Albrecht 79] [Albrecht and Gaffney 83] give a measure of the functionality of the system starting from a description, in natural language, of user's requirements. Thus they provide a technology independent estimate of the size of the final program and are probably the only measure of size that is not related to code.
The measurement of function points is based on identifying and counting the functions that the system has to perform. There are five different function types:
1. *external inputs*
2. *external outputs*
3. *logical internal files*
4. *external interface file*
5. *external inquiry*

Each function identified in the system is then classified to three levels of complexity: simple, average and complex. According to the complexity and the function type, a weight is assigned to each function and all the weights are summed up to give the unadjusted function point count. The final function point count is then obtained by multiplying the unadjusted count by an adjustment factor that expresses the influence of 14 general system characteristics.
Let us now see the definitions of the five function types considering a sample application for the management of human resources, which should store information about all the employees of a firm.

1 An *external input* is a *unique* user data or control input that enters the external boundaries of the application and *adds* or *change* data in a logical internal file. An external input is unique if it has a different *format* or if it requires a *processing logic* different from other external inputs. An example is the insertion of a new employee in the database.
2 An *external output* is a *unique* user data or control output that leaves the external boundary of the application. The definition of a unique external output is analogous to the one for external input. An example is the production of a report containing the list of all the jobs together with the employees to which they are assigned.
3 A *logical internal file* is a logical group of data or control information that is generated, used and maintained by the application. Do not confuse a logical file with its implementation by physical files, there can be more physical files for the same logical one. An example is the group of information related to the employee, his name, address, salary, department, etc.
4 An *external interface file* is a logical group of data or control information that is used by the application but is generated and maintained by another application (it is an internal logical file for the other application). An example is the information relative to the work site of the employees, that can be maintained by an application about estates.
5 An *external inquiry* type is an input/output combination, where an input causes and generates an immediate output. An example is a query on the database of employees in which you search for all the employees that satisfy certain conditions.

Once a function is identified, it is classified according to three levels of complexity by taking into account the number of *data element types* that uses and of *file types referenced*. The classification is done using the following matrix (that is different for each Function Type, this is the one for external inputs):

	1-4 DET	**5-15 DET**	**16 or more DET**
0-1 FTR	Low	Low	Medium
2 FTR	Low	Medium	High
3 or more FTR	Medium	High	High

FTR = File Types Referenced DET = Data Element Types [IFPUG 94]

Then the number of Function Point assigned to the function can be computed on the basis of the level of complexity according to a matrix that provides, depending on the level of complexity and the type of function, the corresponding number of function points.

The final function point count is then obtained by summing up the contribution of all the functions identified and by multiplying the unadjusted count by an adjustment factor that expresses the influence of 14 general system characteristics. They are:

> *data communication, distributed functions, performance, heavily used configuration, transaction rate, on-line data entry, end user efficiency, on-line update, complex processing, reusability, installation ease, operational ease, multiple sites, facilitate change.*

Each characteristic is assigned a degree of influence from 0 to 5 and the final function point count is then given by the equation:

$$FP = \left[\sum_{i=1}^{n} Weight_i \right] \times \left[0.65 + 0.01 \times \sum_{j=1}^{14} DegreeOfInfluenceOfGSC_j \right]$$

where n is the number of identified functions and $Weight_i$ is the weight of function i according to its type and complexity.

[Albrecht and Gaffney 83] showed that function point are highly correlated with work-hours and application size in lines of code and therefore argued that function point can be used effectively for estimating effort.

From a measurement theory point of view the use of the adjustment factor invalidates the measure. The unadjusted count could be considered a reasonable measure of functionality in specification document. The adjustment factor is introduced because the authors wanted function points to have predictive power for effort. But adjusting the function point count with the 14 general system characteristics is like correcting the measure of height of people in such a way that the measure correlates more closely with intelligence ([Fenton 94]). Moreover, empirical studies have shown that the unadjusted count is a better predictor of effort ([Stathis, Jeffrey 93]).

6.2 High-Level Design metrics

[Briand et al. 94] propose a methodology for defining and validating software metrics for high-level designs, either traditional or object oriented. The goal of these metrics is to assess the quality of a software design with respect to its error proneness.

The authors propose four families of metrics but we will describe here only those based on interactions that comprehends metrics for cohesion and coupling.

The high level design of a system is seen as a collection of *modules*. A module is a provider of a computational service and is a collection of *features*, i.e. constants, type, variable and subroutine definitions. It is an object in object oriented systems, but can be present as well in traditional systems. Modules are composed of two

parts: an interface and a body (which may be empty). The interface contains the computational resources that the module makes visible for use to other modules. The body contains the implementation details that are not to be exported. The high-level design of a system consists only in the definitions of the interfaces of modules. A *software parts* is a collections of modules.
Let us now give the definitions of interactions.

Definition 1: *Data declaration-Data declaration (DD) Interaction*
A data declaration A *DD-interacts* with another data declaration B if a change in A's declaration or use may cause the need for a change in B's declaration or use.

Definition 2: *Data declaration-Subroutine (DS) Interaction*
A data declaration *DS-interacts* with a subroutine if it DD-interacts with at least one of its data declarations.

Let us consider now the attribute *cohesion*. It is the extent to which features that are conceptually related belong to the same module. It is desirable to have an high cohesion because otherwise we would have features that depend on each other scattered all over the system, with the result that the software could be more error-prone.
In order to define a metric for cohesion, we have to define first cohesive interactions.

Definition 3: *Cohesive interactions*
The set of cohesive interactions in a module *m* is the union of the sets of DS-interactions and DD-interactions, with the exception of those DD-interactions between a data declaration and a subroutine formal parameter.

The interactions between a data declaration and a subroutine formal parameter are not considered because they are already accounted for by DS-interactions.
The measure proposed for cohesion is the *Ratio of Cohesive Interactions* (RCI) for a Software Part (*sp*):

$$RCI(sp) = \frac{\#CohesiveInteraction(sp)}{\#MaxInteraction(sp)}$$

where the maximum number of cohesive interactions for a module is obtained by connecting every data declaration with every other data declaration and subroutine with which it can interact. The number of cohesive and maximum interactions for a software part (a group of modules) is obtained by summing the numbers for each single module

At high-level design time, only the interactions among features in the interfaces of modules are known, but nothing is known about interactions with features in the body of modules, for example global variables accessed by subroutine bodies. However, some additional information at the end of high-level design may be available. For example, given the interface of a module *m*, the designer have at least a rough idea of which object declared in *m* will be manipulated by a subroutine in *m*'s interface.
Therefore we can have three cases for an interaction. The interaction is known if it is detectable from the high-level design or if it is signaled by the designers, it is unknown if it is not detectable from the high-level design and it is not signaled by the designers. The third case is when the interaction is not detectable from the high-level design but the designers explicitly exclude their existence.
Using this additional information, it is possible to define the metrics

Neutral Ratio of Cohesive Interactions (NRCI)
Unknown interaction are not taken into account

$$NRCI(sp) = \frac{\#CohesiveInteraction(sp)}{\#MaxInteraction(sp) - \#UnknowknInteractions(sp)}$$

Pessimistic Ratio of Cohesive Interactions (PRCI)
Unknown interaction are considered not to be actual interactions

$$PRCI(sp) = \frac{\#CohesiveInteraction(sp)}{\#MaxInteraction(sp)}$$

Optimistic Ratio of Cohesive Interactions (ORCI)
Unknown interaction are considered as if they were known to be actual interactions

$$ORCI(sp) = \frac{\#CohesiveInteraction(sp) + \#UnknownInteractions(sp)}{\#MaxInteraction(sp)}$$

From the definitions of *PRCI(sp), NRCI(sp) and ORCI(sp),* it can be shown that:
$0 \le PRCI(sp) \le NRCI(sp) \le ORCI(sp) \le 1$
ORCI(sp) and *PRCI(sp)* provide the bounds for the admissible range for cohesion and NRCI(sp) takes a value in between.

Let us consider now the attribute *Coupling*.

Definition 4: *Import Coupling*

Import Coupling is the extent to which a software part depends on imported external data declarations.

Definition 5: *Export Coupling*
Export Coupling is the extent to which data declarations of a software part effect the data declarations of other software parts in the system.

It is desirable to have low coupling in both cases so that to reduce the number of interdependencies between different parts of the system that can lead to error. Let us now see the proposed measures for coupling.

Import coupling of a software part *sp* is measured by the number of data declarations external to *sp* that interact with data declarations within *sp*.

Export coupling of a software part *sp* is measured by the number of data declarations within *sp* that interact with data declarations external to *sp*.

As regards validation of the metrics, from a theoretical point of view the authors demonstrates that these metrics satisfy specific properties for cohesion and coupling which ensures that they are defined on a ratio scale. From an empirical point of view, the authors have validated the metrics on a set of NASA projects in order to find out if they are correlated with error proneness. The result of this study is that PRCI and Import Coupling is related to error proneness, while Export Coupling is not.

6.3 Object Oriented Metrics

[Chidamber and Kemerer 94] have defined a suite of metrics for object oriented designs. They define metrics for cohesion, coupling, complexity, depth of inheritance, number of children and response set. For cohesion and coupling, we will underline the differences between these definitions and those previously seen. The metrics proposed are all referred to the individual class and not to the whole system.
Lack of Cohesion in Methods (LCOM): Two methods share an instance variable if they both use it. Let P be the number of pairs of methods that do not share instance variables, let Q be the number of pairs of methods that share instance variables. LCOM is defined as:
LCOM =P-Q if P>Q
=0 otherwise

The interactions considered for cohesion are different from the ones considered in the previous section. Here only interactions between methods are considered and two methods interact if the use at least one common variable, while Briand et al. consider interactions only between data declarations and between data and subroutine declarations. This means that this metric can be collected only after implementation, while the previous one could be measured already on the high-level design.

Coupling Between Object Classes (CBO): for a class, it is the number of other classes to which it is coupled. Two classes are coupled if one of them acts on the other, i.e. methods of one use methods or instance variables of the other.

This differs as well from the previous case when only interactions among data declarations were considered and also this metric can be collected only after implementation.

Weighted methods per Class (WMC): sum of the complexities of the methods of a class. The authors leave open the choice of a metric for complexity, as long as it is defined on a interval scale and can therefore be summed.

Depth of Inheritance Tree of a class (DIT): depth of the class in the inheritance tree. In the case of multiple inheritance, the DIT is the maximum length from the root to the class.

Number Of Children of a class (NOC): number of immediate subclasses subordinated to a node in the hierarchy.

Response Set for a class (RFC): cardinality of the set of methods that can be executed (directly or indirectly) in response to a message received by an object of that class.

The validation of these metrics is weak: they have used Weyuker's properties but not all the properties are satisfied. Property 9 is not satisfied by any of the metrics, and the authors conclude that is the property that is at fault and not the metrics. This property states that when two programs are combined together, the resulting complexity can be bigger than the sum of the initial complexities. This means that when a class is split into two classes, the complexity can decrease. But this is contrary to the experience of many OO designers interviewed by the authors: memory management and run-time error detection is more difficult when there are a large number of classes to deal with. Therefore, dividing classes into more classes can increase complexity, and the authors conclude that Property 9 is not adequate for OO complexity metrics.

The authors have conducted as well an experimental work: they have collected the metrics for two class libraries at two different sites and they represented the data on histograms with the values of the metrics on the x axis and the number of classes on the y axis. From the analysis of the histograms and of statistical

quantities such as Maximum, Minimum and Median value, they derive comments on the design decision taken by the developers.

6.4 Lines of code

Probably the simplest software metric is the number of lines of code. But even for the definition of such a simple metric, we have some problems: it is not clear if we have to count the comments as well, even if they give a big contribution to the understandability of the program, and the declarative parts, that give a contribution to the quality of the program. Moreover some languages allow more than one instruction on the same line. If we count the number of instructions, we are still uncertain about comments and declarations.

Currently the most accepted definition of LOC (Lines Of Code) is given by [Conte et al. 86]:

"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements."

One of the main disadvantages with LOC is that it does not take into account the goodness of the code: if we use LOC to measure productivity, a short well-designed program is "punished" by such a metric.

Another disadvantage is that it does not allow to compare programs written in different languages.

In spite of these problems, lines of code are a widely used metric due to its simplicity, ease of application, inertia of tradition and absence of alternative size measures. Moreover, there are many empirical studies that demonstrates the usefulness of LOC: [Basili 80] found that effort correlated better with LOC that with Halstead's metrics and was at least as good as McCabe's cyclomatic complexity. This and other studies ([Evangelist 83]) demonstrate that LOC is better or at least as good as any other metrics for some attributes of software. Therefore LOC have been used for a variety of tasks in software development: planning, monitoring the progress of projects, predicting (for example, the effort estimating model COCOMO [Bohem, 81]) and they have been used as a baseline for the evaluation in almost all empirical studies on metrics, including function points [Albert and Gaffney 1983].

From the point of view of measurement theory, LOC are a valid metric for the length attribute of a program [Kitchenham et al 95] because the empirical relation "is shorter than" is perfectly represented by the relation < between lines of code:

$$x \text{ "is shorter than" } y \Leftrightarrow LOC(x) < LOC(y)$$

6.5 Halstead's Software Science

It was an attempt by M. Halstead to build a complete theory that could describe the measurable attributes of software starting from a few simple elements [Halstead 75]

It is based on the assumption that a program is made only of operators and operands and that the knowledge of the numbers of distinct and repeated operators and operands is sufficient for determining a number of attributes of software such as program length, volume, level, programming effort. It is interesting to note that, even if the equations provides only estimates, they are exact, not statistical.

6.5.1 Fundamental definitions

Operand: every variable or constant present in the program

Operator: every symbol or combination of symbols that influences the value or order of an operand. Puntuation marks, arithmetic symbols (such as +,-,* and /), keywords (such as if, while, do, etc.), special symbols (such as :=, braces, parenthesis, ==, !=) and function names are operators.

Some attributes are considered fundamental and used to derive all the other attributes of the model:

η_1 number of distinct operators

η_2 number of distinct operands

N_1 total number of times that the operators appear in the program

N_2 total number of times that the operands appear in the program

η_2^* number of conceptually distinct input/output operands

This last quantity is the number of parameters on which the program operates if we think about it as a function that taking some values as input, produces some values as output.

6.5.2 The equations

The *length* N of a program is the total number of symbols in the program and is given by

$$N=N_1+N_2$$

This value can be obtained only when the program has been completed, therefore Halstead introduces an estimator of the length N

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

This estimator is useful because η_1 and η_2 can be determined already at the beginning of the coding phase. According to the Software Science, the more the real value is close to the estimated value, the better is the quality of the program. This has been confirmed by some experimental observations [Halstead 75], comparing a subjective classification of programs with the differences between the length and its estimator.

Another quantity considered is the *volume* of a program, defined as the number of bits necessary to represent the program. If we define the *vocabulary* of a program as

$$\eta = \eta_1 + \eta_2$$

then the number of bits necessary to represent each element is $\log_2 \eta$, therefore to represent the whole program we have

$$V = N \log_2 \eta$$

The *potential volume* V^* is defined as a baseline to the volume and it is the volume of a program in a programming language in which the functionality that the program realizes is one of the predefined functionalities of the language. In this case, the total number of operands is given by η_2^* and the operators are 2, the name of the predefined procedure and an operator to separate the arguments, therefore

$$V^* = (\eta_2^* + 2) \log_2 (\eta_2^* + 2)$$

Using V and V^*, Halstead defines the indicator L of the *level* of the program:

$$L = V^*/V$$

Moreover he gives an estimator $\hat{L}$ for L that does not contain η_2^*, since this quantity is often difficult or not possible to estimate

$$\hat{L} = 2\eta_2/(\eta_1 N_2)$$

The inverse of this quantity is the *difficulty of the program*: a program is more difficult when its level decreases, that is when its volume grows with respect to the potential volume.

From the difficulty of a program, Halstead derives the *effort* required to write a program, defined as the number of mental discriminations used in writing the code. Supposing that each of the N choices among the η symbols of the vocabulary is done using a selection algorithm whose complexity is equal to one of binary search, then the total number of choice operations is $N \log_2 \eta$, that is the volume V. Supposing also that the number of mental discriminations for each of these choice operations is proportional to the difficulty of the program, we have:

$$E = V/L = V^2/V^*$$

From the effort, Halstead derives the *time* needed to write a program. He assumes that the number S of mental discriminations per second that can be done by a human is known, therefore:

$$\hat{T} = E / S$$

in which usually the value of the estimator for L is used instead of the real value. As regards S, it estimated to be among 5 and 20 and to be 18 for programming.

6.5.3 Critical discussion

The Software Science of Halstead is an attempt to give a set of simple equations that, starting from very simple quantities, give an account of many different and very complex attributes of code.

From a measurement theory point of view, the length, the volume and the potential volume of a program are valid measures because the empirical relation system is clearly defined for them and their measures respect the representation condition. It is not demonstrated instead that the estimator for the length satisfies the representation condition. No empirical relation system is defined for the level, while for effort and time the empirical relation system is defined but the representation condition is not verified.

Even not considering measurement theory, these equations can be criticized because the way in which they are found is not scientific. They are based on assumptions that neither have a sound demonstration neither are drawn from experimental data. For example, the estimator of the length is based only on qualitative arguments. The effort equation is based on the assumption that the chosen algorithm has the complexity of a binary search but this is not confirmed by any experimental or theoretic evidence. Not even the assumption that the effort is proportional to difficulty has any sound motivation.

From a practical point of view, there is no general agreement among researchers on how to count operators and operands. For example, it is not clear if operators that appear in pairs (like opening and closing parenthesis "()", "begin-end") have to be counted as one or two. The counting scheme is language dependent, and software science counts are very sensitive to the language [Elshoff 78] therefore it is not possible to compare programs written in different languages. This is due to the fact that Halstead developed his theory with algorithms in mind and not programs.

In reality, the values obtained with these equations very rarely match with data measured empirically: for example, in the case of the length, two programs that do very different things, but are built with the same number of distinct operators and operands, are very unlikely to have the same length. The mere knowledge of the

number of distinct operand and operators has too little information inside about semantics to give account for the whole variety of programs that one can write.
Since these equations cannot give exact values for every single case, their validity is referred to the medium case, that can be represented by a champion of real programs. Unfortunately, this is a big limit because they are not built using the methods of statistics and therefore do not have the interesting properties of statistics.
In spite of all these problems, Halstead metrics have gained a wide attention because they constitute the first attempt to define a metric for software. Even if they have not a strong theoretical base and have not been scientifically validated using measurement theory, they have started the discussion and have been used for years as a comparison for the new metrics.

6.6 McCabe's Cyclomatic Number

Differently from Halstead, [McCabe 76] does not try to build a comprehensive model of software but concentrates only on the complexity attribute. More specifically he concentrates on control flow complexity and does not take into account the contribute to complexity that derives from data.
McCabe's approach does not start with a rigorous definition of complexity but proposes his metric as an operating definition of control flow complexity.
A program control flow can be represented by a graph which has a unique entry node and exit node, and in which all nodes are reachable from the entry and the exit is reachable from all nodes.
His idea is to measure the complexity by considering the number of paths in the control graph of the program. But even for simple programs, if they contain at least one cycle, the number of paths is infinite. Therefore he considers only the number of *independent paths*: these are complete paths, (paths that go from the starting node to the end node of the graph), such that their linear combinations can produce all the set of complete path of a program (see Figure 4)

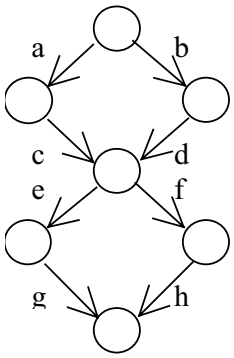

Figure 4

In this graph there are four possible paths: aceg, acfh, bdeg, bdfh. We can represent each path as a vector with eight positions, one for each arc, containing the number of times that the arc is present in the path. In this case we would see that, of the four vectors, only three are independent, while the fourth is always a linear combination of the other three. Therefore the number of independent paths is 3. In graph theory can be demonstrated that in a strongly connected graph (one in which each node can be reached from any other node) the number of independent paths is given by

$$e-n+1$$

where e is the number of nodes and n is the number of arcs.

However we cannot apply this formula directly to the control flow graph of a program because it could be not strongly connected. But we can make it such by adding an arc from the end node to the start node of the program. In this way we increase by one the number of arcs and therefore the number of independent paths (as a function of the original graph) is given by

$$v(G)=e-n+2$$

and this is the cyclomatic number as defined by McCabe.

This formula has an interesting simplification: we can calculate the cyclomatic number only by knowing the number of choice points d in the program (Mills' theorem):

$$v(G)=d+1$$

We assume that a k-way decision point contributes for k-1 choice points.

Moreover, the definition of cyclomatic number can be extended to programs with procedures inside. These are represented as separate flow graphs, they can be called by any node of the main program but the connections are not made explicit. The cyclomatic number of the whole graph is then given by the sum of the numbers of each graph. It is easy to demonstrate that, if p is the number of graphs and e and n are referred to the whole graph, the cyclomatic number is given by

$$v(G)=e-n+2p$$

and Mills' theorem becomes

$$v(G)=d+p$$

6.7 Critical Discussion

With respect to software science, the cyclomatic number has a much more limited scope, but it is a more accurate study of control flow complexity. It has strong mathematical foundations, unlike software science, and it seems to capture at least a part of the intuitive meaning of control flow complexity. Moreover, McCabe has, partially, validated experimentally his metric by showing that there is a degree of correlation between the cyclomatic number and some quantities that surely influence control flow complexity, like reliability [McCabe 76]. From this experimental work, McCabe has derived the empirical rule that the cyclomatic number of a module should not be bigger than 10.

However, if we analyze the metric from the point of view of measurement theory, we have seen the demonstration by Fenton that it is not possible to define a single metric for control flow complexity and therefore the cyclomatic number as a measure of that attribute is not valid.

Even if a single metric for control flow complexity (and, as a consequence, of complexity in general) does not exist, we can still measure consistently single aspects of complexity, such as the number of independent paths, for which the cyclomatic number is a valid measure.

7. Conclusions

In this report we have tried to answer the basic questions about software metrics: why measuring, what to measure, how to measure and when to measure. We have distinguished the two main uses of measurement: assessment and prediction. We have analyzed the attributes of software, distinguishing attributes of process, product or resources. Among all the attributes, complexity is probably the most important one and it comprehends many different aspects of software. Structural complexity is the kind of complexity most studied because it is more objectively measurable. Weyuker has proposed a set of axioms that measures for structural complexity should satisfy.

The representational theory of measurement should be the base of any kind of measurement, included that of software. It defines the measure as a mapping between an empirical relation system and a formal one and it gives the representation condition as a necessary and sufficient condition for a measure to

be valid. Using measurement theory, Fenton has demonstrated that a single measure for complexity cannot exist and that Weyuker axioms are inconsistent.
We have described in detail six metrics, chosen among the ones most widely known and used. They are relative to different phases of software development. In the requirements phase, we can use Function Points to measure the functionality starting from the user requirements.
In the high-level design phase, the suite of metrics by Briand et al. can be used: we have concentrated on measures for cohesion and coupling, which are important attributes of a design.
For the coding phase, we have described the suite of metrics for OO systems by Chidamber and Kemerer, whose metrics for cohesion and coupling are based on a notion of interaction different from the one by Briand et al. Lines of Code is the oldest and the simplest measure of program size. Halstead's software science was an early attempt to give simple prediction models for important attributes such as volume and effort. Lastly, McCabe's cyclomatic number is a measure of program complexity based on the number of independent paths in the control flow graph of the program.
All of these metrics, and most of the metrics defined in the literature, have not been validated using the theory of measurement. Some of them have been validated by showing that they are correlated with other metrics: Function Points have a good correlation with size and effort, while Briand et alii's Pessimistic Cohesion Ratio and Import coupling are correlated with error proneness. Chidamber and Kemerer have validated their metrics using Weyuker's properties. These kinds of weaker validation are very diffused in the literature but in recent time the need for stronger forms of validation have come out. While in the past the focus in research was on inventing new metrics, now the focus is more on measurement theory, in particular on the definition of new validation frameworks or of new set of axioms.

Bibliography

[Albrecht 79] A. Albrecht: "Measuring application development productivity", in *Proc. Joint SHARE/GUIDE/IBM Applications Development Symposium*, Monterey, CA, 1979

[Albrecht and Gaffney 83] A. Albrecht and J. Gaffney: *Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation*; in IEEE Trans. Software Eng., 9(6), 1983, pp. 639-648

[Basili 80] V. Basili, Qualitative Software Complexity Models: a Summary, in *Tutorial on Models and Methods for Software Management and Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 1980

[Bohem, 81] B. Bohem, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, 1981

[Briand et al 94] L. Briand, S. Morasca, V. Basili, *Defining and Validating High-Level Design Metrics*, Tech. Rep. CS TR-3301, University of Maryland, 1994.

[Briand et al. 96] L. Briand, S. Morasca, V. Basili, Property-Based Software Engineering Measurement, IEEE Trans. Software Eng. 22(1), 1996, pp. 68-85.

[Conte et al. 86] S. Conte, H. Dunsmore, V. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park, CA.

[Chidameber, Kemerer 94] S. Chidamber, C. Kemerer, *A Metrics Suite for Object Oriented Design*, IEEE Trans. Software Eng., 20/6), 1994, pp. 263-265

[Elshoff 78] J. Elshoff, An Investigation Into The Effects of the Counting Method Used on Software Science Measurement, ACM SIGPLAN Notices, 13(2), 1978, pp. 30-45

[Evangelist 83] M. Evangelist, *Software Complexity Metric Sensitivity to Program Structuring Rules*, Journal of Systems and Software, 3(3), 1983, pp. 231-243.

[Fenton 91] N. Fenton, *Software Metrics: A Rigorous Approach*, Chapman and Hall, London, 1991

[Fenton 94] N. Fenton, Software Measurement: a Necessary Scientific Basis, IEEE Trans. Software Eng., 20, 1994, pp. 199-206.

[Henderson-Sellers 96] B. Henderson-Sellers, Object Oriented Metrics: Measures of Complexity, Prentice Hall, Upper Saddle River, NJ, 1996

[IFPUG 94] International Function Points User Group, *Function Point: Manuale sulle Regole del Conteggio*, *Versione 4.0*, 1994.

[Kitchenham et al 95] B. Kitchenham, S. Pfleeger, N. Fenton, *Towards a Framework for Software Measurement Validation*, IEEE Trans. Software Eng., 21(12), 1995, pp. 929-944.

[McCabe 76] T. McCabe, *A Complexity Measure*, IEEE Trans. Software Eng., 2(4), 1976, pp. 308-320.

[Morasca 95] S. Morasca, *Software Measurement: State of the Art and Related Issues*, slides from the School of the Italian Group of Informatics Engineering, Rovereto, Italy, September 1995.

[Stathis, Jeffrey 93] J. Stathis, D. Jeffrey, *An Empirical Study of Albrecht's Function Points, in Measurement for Improved IT management*, Proc. First Australian Conference on Software Metrics, ACOSM 93, Sydney, 1993, pp. 96-117.

[Weyuker 88] E. Weyuker, *Evaluating Software Complexity Measures*, IEEE Trans. Software Eng., 14(9), 1988, pp. 1357-1365.

[Zuse 90] H. Zuse, *Software Complexity: Measures and Methods*, Walter de Gruyter, Berlin, 1990