SOFTWARE MEASUREMENT

SANDRO MORASCA
Università dell'Insubria
Dipartimento di Scienze Chimiche, Fisiche e Matematiche
Sede di Como
Via Valleggio 11
Como, I-22100, Italy
Email: morasca@uninsubria.it

This article provides an overview of the basic concepts and state of the art of software measurement. Software measurement is an emerging field of software engineering, since it may provide support for planning, controlling, and improving the software development process, as needed in any industrial development process. Due to the human-intensive nature of software development and its relative novelty, some aspects of software measurement are probably closer to measurement for the social sciences than measurement for the hard sciences. Therefore, software measurement faces a number of challenges whose solution requires both innovative techniques and borrowings from other disciplines. Over the years, a number of techniques and measures have been proposed and assessed via theoretical and empirical analyses. This shows the theoretical and practical interest of the software measurement field, which is constantly evolving to provide new, better techniques to support existing and more recent software engineering development methods.

Keywords: software measurement, software metrics, theoretical validation, empirical validation

1. Introduction

Measurement permeates everyday life and is an essential part in every scientific and engineering discipline. Measurement allows the acquisition of information that can be used for developing theories and models, and devising, assessing, and using methods and techniques. Practical application of engineering in the industry would not have been possible without measurement, which allows and supports

- production planning, from a qualitative and quantitative viewpoint,
- production monitoring and control, from a qualitative and quantitative viewpoint,
- decision making,
- cost/benefit analysis, especially when new techniques are proposed or introduced,
- post-mortem analysis of projects,
- learning from experience.

Like in all other engineering branches, industrial software development requires the application of software engineering methods and techniques that are effective, i.e., they allow software organizations to develop quality software products, and efficient, i.e., they allow software organizations to optimize the use of their production resources.

However, software engineering differs from other engineering disciplines in a number of aspects that have important consequences on software measurement. First, software engineering is a young discipline, so its theories, methods, models, and techniques still need to be fully developed and assessed. Other engineering branches rely on older, well-consolidated scientific disciplines. These disciplines have developed a large number of deep mathematical models and theories over the centuries and have long identified the important concepts (e.g., length, mass) that need to be studied and have long developed suitable tools to measure them. Second, Software Engineering is a very human-intensive discipline, while other engineering branches are based on the so-called hard sciences (e.g., Physics, Chemistry). Therefore, some aspects of software measurement are more similar to measurement in the social sciences than measurement in the hard sciences. For instance, in a number of software measurement applications, repeatability of results may not be achieved. As a consequence, one cannot expect software engineering measurement models and theories to be necessarily of the same nature, precision, and accuracy as those developed for more traditional branches of engineering. Third, there are several different types of software development, for different application areas and purposes, so, software measurement models may not be valid on a general basis, like those used in other engineering branches.

However, the very nature of software engineering makes measurement a necessity, because more rigorous methods for production planning, monitoring, and control are needed, otherwise the amount of risk of software projects may become excessive, and software production may easily get out of industrial control. This would produce obvious damages to both software producers (e.g., higher costs, schedule slippage) and users (e.g., poor quality products, late product delivery, high prices). To be effective and make good use of the resources devoted to it, software measurement should address important development issues, i.e., it should be carried out within a precise goal of industrial interest. In this context, software measurement may serve several purposes, depending on the level of knowledge about a process of product. Here, we list some of these purposes, from one that can be used when a limited or no amount of knowledge is available to one that requires an extensive amount of knowledge.

- Characterization, i.e., the gathering of information about some characteristic of software processes and products, with the goal of acquiring a better idea of ``what's going on."
- Tracking, i.e., the (possibly constant and regular) acquisition of information on some characteristic of software processes and products over time, to understand if those characteristics are under control in on-going projects.
- Evaluation, i.e., judging some characteristic of a software process or product, for instance based on historical data in the same development environment or data available from external sources.
- Prediction, i.e., identifying a cause-effect relationship among product and process characteristics.
- Improvement, i.e., using a cause-effect relationship to identify parts of the process or product that can be changed to obtain positive effects on some

characteristic of interest, and collecting data after the changes have been made to confirm or disconfirm whether the effect was positive and assess its extent.

Therefore, the goal of software measurement is certainly not limited to deriving measures. In addition to the above practical goals, one may say that, from a more abstract point of view, the goal of software measurement is to build and validate hypotheses and increase the body of knowledge about software engineering. This body of knowledge can be used to understand, monitor, control, and improve software processes and products. Therefore, building measures is a necessary part of measurement, but not its final goal.

Software measurement poses a number of challenges, from both a theoretical and practical points of view. To face these challenges, we can use a number of techniques that have been developed over the years and/or have been borrowed from other fields.

First, we need to identify, characterize, and measure the characteristics of software processes and products that are believed to be relevant and should be studied. This is very different from other engineering branches, where researchers and practitioners directly use measures without further thought. In those disciplines, there no longer is a debate on what the relevant characteristics are, what their properties are, and how to measure these characteristics. In software engineering measurement, instead, we still need to reach that stage. There is not as much intuition about software product and process characteristics (e.g., software cohesion or complexity) as there is about the important characteristics of other disciplines. Therefore, it is important that we make sure that we are measuring the right thing, i.e., it is important to define measures that truly quantify the characteristic they purport to measure. This step—called theoretical validation—is a difficult one, in that it involves formalizing intuitive ideas around which there is limited consensus. To this end, one can use Measurement Theory, which has been developed in the social sciences mainly in the last 60 years, or property-based approaches, which have been used in Mathematics for a long time.

Second, we need to show that measuring these characteristics is really useful, via the so-called empirical validation of measures. For instance, we need to show if and to what extent these characteristics influence other characteristics of industrial interest, such as product reliability or process cost. It is worthwhile to measure them and use them to guide the software development process only if they have a sufficiently large impact. To this end, experiments must be carried out and threats to their internal and external validity must be carefully studied.

In addition, the identification and assessment of measures may not be valid in general. Nothing guarantees that measures that are valid and useful in one context and for some specified goal are as valid and useful for another context and goal. Goal-oriented frameworks that have been defined for software measurement can be used.

Before illustrating the various aspects of software measurement, we would like to explain that the term ``metric'' has been often used instead of ``measure'' in the software measurement field in the past. As it has been pointed out, ``metric'' has a more specialized meaning, i.e., distance, while ``measure'' is the general term. Therefore, we use ``measure'' in the remainder of this article.

The remainder of this article is organized as follows. Section 2 describes a framework for goal-oriented measurement (the Goal/Question/Metric paradigm). Sections 3 and 4 introduce the basic concepts of software measurement. Sections 5 and 6 illustrate how Measurement Theory and axiomatic approaches can be used to carry out the so-called theoretical validation of software measures, i.e., to show that a measure actually quantifies the attribute it purports to measure. Section 7 describes some of the measures that have been defined for the intrinsic attributes (e.g., size, structural complexity) of the artifacts produces during software development. Section 8 concisely reports on external software attributes (e.g., reliability, maintainability), i.e., those that refer to the way software relates to its development or operational environment. Process attributes are discussed in Section 9. Remarks on the practical application of software measurement are in Section 10. Possible future developments are discussed in Section 11.

Good surveys of the state of the art and on-going research can be found in [1, 2].

2. Goal-oriented Measurement

It is fundamental that all measurement activities be carried out in the context of a well-defined measurement goal. In turn, the measurement goal should be clearly connected with an industrial goal, so the measurement program responds to a software organization's needs. The Goal/Question/Metric (GQM) paradigm [3, 4] provides a framework for deriving measures from measurement goals. The idea is to define a measurement goal, with five dimensions, as follows:

- *Object of Study*: the entity or set of entities that should be studied, e.g., a software specification, or a testing process;
- *Purpose*: the reason/the type of result that should be obtained: e.g., one may want to carry out/obtain a characterization, evaluation, prediction, or improvement;
- *Quality Focus*: the attribute or set of attributes that should be studied, e.g., size (for the software specification, or effectiveness (for the testing process);
- *Point of View*: the person or organization for whose benefit measurement is carried out, e.g., the designers (for the software specification), or the testers (for the testing process)
- *Environment*: the context (e.g., the specific project or environment) in which measurement is carried out.

The following is an example of a GQM goal:

Analyze the testing process (object of study) *for the purpose of* evaluation (purpose) *with respect to* the effectiveness of causing failures (quality focus) *from the point of view of* the testing team (point of view) *in the environment of* project X (environment).

GQM goals help clarify what needs to be studied, why, and where. Based on the goal, the relevant attributes are identified via a set of questions (an intermediate document between the goal and the questions, the Abstraction Sheet, has been recently

introduced to provide a higher-level view of the questions). As a simple example, a question related to the study of testing effectiveness may ask: How much is defect density? Each question is then refined into measures that can be collected on the field. For instance, defect density may be defined as the ratio of the number of defects found to the number of lines of code. Figure 1 shows this top-down refinement of goals into measures. Figure 1 also shows that several measurement goals may be pursued at the same time, and questions and measures may be reused across goals, thus decreasing the effort for adding further goals to an existing set of goals, questions, and measures. Conversely, interpretation of results proceeds bottom-up, i.e., the measures collected are used and interpreted in the context and for the objectives that have been initially defined. The GQM paradigm has been successfully used in many industrial environments to increase the knowledge of software organizations about their own practices and lay the quantitative foundations for improvement of software processes and products.

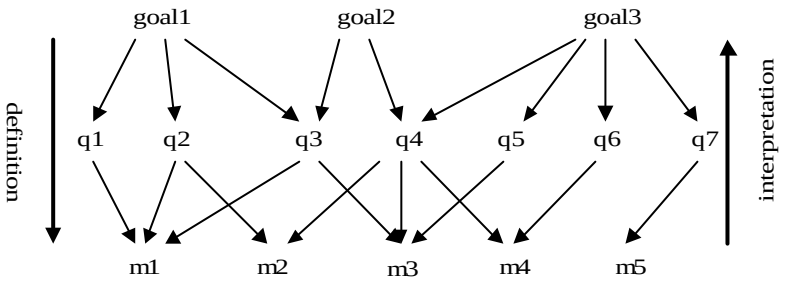

Fig. 1. A sample GQM.

The GQM paradigm is a part of an organized approach to the improvement of software products and processes, the Quality Improvement Paradigm (QIP), which can be concisely described as an instantiation of the scientific method tailored for the needs of Software Engineering. The QIP is based on the idea that improvement can be achieved and quantified via the acquisition of knowledge on software processes and products over time. The Experience Factory (EF) is another central aspect of the QIP. The EF is the organizational unit that collects, stores, analyzes, generalizes, and tailors information from software development projects, so it can be used in future ones.

3. Entities and Attributes

We now introduce the basic concepts of measurement. First, we identify the object of measurement. To this end, measurement is based on the following two concepts [1].

Entity. An entity may be a physical object (e.g., a program), an event that occurs at a specified instant (e.g., a milestone) or an action that spans over a time interval (e.g., the testing phase of a software project).

Attribute. An attribute is a characteristic or property of an entity (e.g., the size of a program, the time required during testing).

Both the entity and the attribute to be measured must be specified, because measurement should not be used for entities or attributes alone. For instance, it does not make sense to ``measure a program,'' since the attribute to be measured is not specified (it could be size, complexity, maintainability, etc.), or to ``measure size,'' since the entity whose size is to be measured is not specified (it could be a specification, a program, a development team, etc.). Instead, one can ``measure the size of a program.''

Entities in software measurement can be divided into two categories: products and processes. Product and process entities may be of different kinds. For instance, requirements, specifications, designs, code, test sets are all product entities and their parts are product entities as well. On the other hand, single phases, activities, and resources used during a project are process entities. Product and process entities have specific attributes. For instance, product attributes include size, complexity, cohesion, coupling, reliability, etc. Process attributes include time, effort, cost, etc.

Attributes are usually divided into internal and external attributes. An internal attribute of an entity depends only on the entity. For instance, the size of a program may be considered an internal attribute, since it depends only on the program. An external attribute of an entity depends on the entity and its context. For instance, the reliability of a program depends on both the program and the environment in which the program is used. External attributes are usually the ones whose measures have industrial interest. For instance, it is important to measure the reliability of a program during its operational use, so as to assess whether its quality is sufficiently high. However, external attributes are usually difficult to measure directly, since they depend on the environment of the entity. Internal product attributes are easier to measure, but their measurement is seldom interesting *per se*, at least from a practical point of view. Internal product attributes are measured because they are believed to influence the external attributes (e.g., coupling is believed to influence maintainability) or process attributes (e.g., program complexity is believed to influence cost).

4. Measurement and Measure

We introduce a distinction between measurement and measure, as follows.

Measurement. Measurement is the process through which values are assigned to attributes of entities of the real world.

Measure. A measure is the result of the measurement process, so it is the assignment of a value to an entity with the goal of characterizing a specified attribute.

Therefore, a measure is not just a value, but it is a function that associates a value with an entity. In addition, the notion of value is by no means restricted to real or integer numbers, i.e., a measure may associate a symbol with an entity. For instance, a measure for the size of programs may associate the values ``small'', ``medium,'' and

``large'' with programs. These values cannot be used for arithmetic or algebraic manipulations, but they can be used to obtain useful results through a variety of statistical techniques.

Because of the distinction between internal and external attributes, measures for internal attributes of an entity can be computed based only on the knowledge of the entity, while measures for external attributes also require the knowledge of the context of the entity.

Not all functions that associate a value with an entity are sensible measures for an attribute of that entity. For instance, if we intuitively rank a program P' as longer than another program P'', we expect that a sensible measure of program size gives program P' a value greater than the value it gives program P''.

Measures that make sense on an intuitive level are an obvious precondition in every measurement application. The measures used in many scientific fields (especially in the hard sciences) are usually taken for granted, i.e., they are considered intuitively valid for the attributes they purport to measure. For instance, there is no debate about the measures used to quantify the volume of an object. The reason is that there is a consolidated consensus about which measures can be considered intuitively valid. This consensus has been built over centuries and is so well established that the question hardly ever arises whether a measure is a sensible one for an attribute. However, this is not the case in software engineering measurement. Because of the novelty of the field, a sufficient degree of consensus still needs to be reached about which measures make sense on an intuitive level. Moreover, there is no widespread agreement on the attributes themselves, i.e., how can we define attributes unambiguously? which attributes are really relevant?

These questions are important from an engineering point of view, because methods and techniques are devised with reference to attributes of software entities. For instance, it is commonly accepted that software systems should be composed of modules (entities) with high levels of inner cohesion (attribute) and low levels of coupling (attribute) among them. These software systems (entities) are believed to have, for instance, a high degree of maintainability (attribute). This is one of the reasons why object-oriented programming is believed to lead to better software systems than other techniques. However, this hypothesis (i.e., there is an influence of module cohesion and coupling on the maintainability of a software system) cannot undergo a real scientific examination until unambiguous definitions are provided for cohesion, coupling, and maintainability, and until measures are defined that are consistent with these definitions when applied to software modules and systems. This is not to say that module cohesion and coupling have no impact on software system maintainability or that object-oriented programming has not been beneficial in software development. However, in a sound engineering approach we need to ascertain

- whether the evidence is sufficient for us to draw reliable conclusions (e.g., is the evidence only anecdotal or do statistical procedures such as the test of hypotheses confirm the hypothesis?)

- the absence of other influencing factors (e.g., has maintainability improved because of the adoption of object-oriented programming or because of the improvement in the education of programmers or the adoption of CASE tools?)
- the extent of the impact of the influencing attributes on the influenced attribute (e.g., does module cohesion influence software maintainability more than module coupling does? what is the impact of a variation in module coupling on maintainability?)

If we had information on these three points above, we would be able to use software techniques more effectively.

We now describe two approaches (Representational Measurement Theory and property-based approaches) that help identify those measures that make sense. It is to be borne in mind that, in both cases, we are actually modeling intuition, and intuition may vary across different people. For instance, one person may not agree with another person on how programs should be ranked according to their size. However, the value added by either approach is to spell out one's own intuition in mathematical, unambiguous terms, which can be used as a basis for discussion about the measures for an attribute and, eventually, to reach widespread consensus.

5. Representational Measurement Theory

Representational Measurement Theory [5] (see also [1, 6]) formalizes the ``intuitive,'' empirical knowledge about an attribute of a set of entities and the ``quantitative,'' numerical knowledge about the attribute. The intuitive knowledge is captured via the so-called empirical relational system (described in Section 5.1), and the quantitative knowledge via the so-called numerical relational system (Section 5.2). Both the empirical and the numerical relational systems are built by means of set algebra. A measure (Section 5.3) links the empirical relational system with the numerical relational system in such a way that no inconsistencies are possible, as formalized by the Representation Condition. In general, many measures may exist that quantify equally well one's intuition about an attribute of an entity (e.g., weight can be measured in kilograms, grams, pounds, ounces, etc.). The relationships among the admissible measures for an attribute of an entity are illustrated in Section 5.4. As we will see, this leads to classifying measures into different categories, as described in Section 5.5. ``Objectivity'' and ``subjectivity'' of measures are discussed in Section 5.6.

5.1. Empirical Relational System.

An empirical relational system for an attribute of a set of entities is defined through the concepts of set of entities, relations among entities, and operations among entities. Therefore, an empirical relational system is defined as an ordered tuple $ERS = \langle E, R_1, \ldots, R_n, o_1, \ldots, o_m \rangle$, as we now explain.

- E is the set of entities, i.e., the set of objects of interest that are subject to measurement with respect to that attribute.

- $R_1,...,R_n$ are empirical relations: each empirical relation R_i has an *arity* n_i, so $R_i \subseteq E^{n_i}$, i.e., R_i is a subset of the cartesian product of the set of entities $E \times E \times ... \times E$ n_i times.
- $o_1,...,o_m$ are binary operations among entities: each binary operation o_j is a function $o_j : E \times E \to E$. Therefore, its result is an entity, so we have $e_3 = e_1 o_j e_2$ (infix notation is usually used for these operations). As an additional assumption, all binary operations o_j's are closed, i.e., they are defined for any pair of entities $\langle e_1, e_2 \rangle$.

As an example, suppose that we want to study the size (attribute) of program segments (set of entities). A program segment is defined as a sequence of statements. To characterize the size attribute, we may define the binary relation ``longer than,'' i.e., $longer\_than \subseteq E \times E$. Therefore, given two program segments e_1 and e_2, we may have $\langle e_1, e_2 \rangle \in longer\_than$ (i.e., our intuitive knowledge is that e_1 is longer than e_2) or $\langle e_1, e_2 \rangle \notin longer\_than$ (i.e., our intuitive knowledge is that e_1 is not longer than e_2). If we were interested in another attribute, e.g., complexity, we would characterize the complexity attribute via a different relation, e.g., $more\_complex\_than \subseteq E \times E$. A binary operation may specify how program segments may be built based on other program segments, e.g., by concatenating program segments. For instance, $e_3 = e_1 \oplus e_2$ may represent the fact that program segment e_3 is built by concatenating program segments e_1 and e_2.

The empirical relations do not involve any values or numbers. We are not comparing the values obtained by measuring e_1's and e_2's sizes, but we are just stating our intuitive understanding and knowledge on e_1's and e_2's sizes and that e_3 is the concatenation of e_1 and e_2. Since this knowledge is intuitive, empirical relational systems are built in a somewhat subjective way, as intuition may very well vary across different individuals.

5.2. Numerical Relational System.

The intuitive knowledge of the empirical relational system is translated into the numerical relational system, which predicates about values. A numerical relational system is defined as an ordered tuple $NRS = \langle V, S_1, ..., S_n, \bullet_1, ..., \bullet_m \rangle$, as we show.

- V is the set of values that can be obtained as results of measures.
- $S_1,...,S_n$ are numerical relations: each numerical relation S_i has the same *arity* n_i of the empirical relation R_i, so $S_i \subseteq V^{n_i}$, i.e., S_i is a subset of the n_i-times cartesian product of the set of values.

- $\bullet_1,...,\bullet_m$ are binary operations among values: each binary operation $\bullet_j$ is a function $\bullet_j : V \times V \to V$. Therefore, its result is a value, so we have $v_3 = v_1 \bullet_j v_2$ (infix notation is usually used for these operations). As an additional assumption, all binary operations $\bullet_j$'s are closed, i.e., they are defined for any pair of values $\langle v_1, v_2 \rangle$.

For instance, the set V may be the set of nonnegative integer numbers. A binary relation may be ``greater than," i.e., $>$, so $> \subseteq V \times V$. A binary operation may be the sum between two integer values, i.e., $v_3 = v_1 + v_2$. Therefore, the numerical relational system in itself does not describe anything about the entities and the attribute.

5.3. Measure

The link from the empirical relational system and the numerical relational system is provided by the definition of measure, which associates entities and values, and scale, which associate the elements of the tuple of the empirical relational system with elements of the numerical relational system. A measure is a function $m : E \to V$ that associates a value with each entity.

As defined above, a measure establishes a link between the set of entities and the set of values, regardless of the relations and operations in the empirical and numerical relational system. Therefore, a measure for the size of program segments may be inconsistent with our intuitive knowledge about size as described by the relation $longer\_than$. For instance, we may have three program segments e_1, e_2, and e_3 such that $\langle e_1, e_2 \rangle \in longer\_than$, $\langle e_2, e_3 \rangle \in longer\_than$, $m(e_1) < m(e_2)$, and $m(e_2) > m(e_3)$. This shows that not all measures are sensible ways for quantifying intuitive knowledge. The Representation Condition places a constraint on measures, so they do not exhibit this kind of counterintuitive behavior.

Representation Condition. A measure $m : E \to V$ must satisfy these conditions

$$\forall i \in 1..n \forall \langle e_1,..., e_{n_i} \rangle \in E^{n_i} \left(R_i(e_1,..., e_{n_i}) \Leftrightarrow S_i(m(e_1),..., m(e_{n_i})) \right)$$
$$\forall j \in 1..m \forall \langle e_1, e_2 \rangle \in E \times E \left(m(e_1 o_j e_2) = m(e_1) \bullet_j m(e_2) \right)$$

The first condition requires that a tuple of entities be in the relation R_i if and only if the tuple of measures computed on those entities is in the relation S_i that corresponds to R_i. Therefore, if the relation $>$ is the one that corresponds to $longer\_than$ we have $\langle e_1, e_2 \rangle \in longer\_than$ if and only if $m(e_1) > m(e_2)$, as one would intuitively expect. The second condition requires that the measure of an entity

obtained with the binary operation o_j from two entities be obtained by computing the corresponding binary operation $\bullet_j$ on the measures computed on those two entities.

Scale. A scale is a triple $\langle ERS, NRS, m \rangle$, where $ERS = (E, R_1, ..., R_n, o_1, ..., o_m)$ is an empirical relational system, $NRS = (V, S_1, ..., S_n, \bullet_1, ..., \bullet_m)$ is a numerical relational system, and $m : E \rightarrow V$ is a measure that satisfies the Representation Condition.

For simplicity, we also refer to *m* as a scale in what follows.

5.4. Uniqueness of a Scale

Given an empirical relational system and a numerical relational system, two issues naturally arise, i.e., existence and uniqueness of a scale that maps the empirical relational system into the numerical relational system. We do not deal with the existence issue here, so we assume that a scale exists that links an empirical relational system and a numerical relational system. As for uniqueness, many different scales can be used given an empirical relational system and a numerical relational system. This is well known in everyday life or scientific or engineering disciplines. For instance, the distance between two points may be measured with different scales, e.g., kilometers, centimeters, miles, etc. All of these are legitimate scales, i.e., they satisfy the representation condition, so it does not really matter which scale we choose. Using one instead of another is just a matter of convenience, since there exists a transformation of scale (multiplication by a suitable proportionality coefficient) from a scale to another scale. However, multiplication by a suitable proportionality coefficient is also the only kind of transformation that can be applied to a distance scale to obtain another distance scale. Other transformations would not lead to a distance scale, since they would ``distort'' the original distance scale. For instance, given a distance scale *m*, using $m' = m^2$ as a distance measure would distort the original distance scale, so we cannot use m^2 as a legitimate distance measure.

In general, this leads to the notion of admissible transformation, i.e., a transformation that leads from one scale to another scale.

Admissible Transformation. Given a scale $\langle ERS, NRS, m \rangle$, a transformation f is admissible if $\langle ERS, NRS, m' \rangle$ is a scale, where $m' = f \circ m$ is the composition of *f* and *m*.

As a different example, suppose that we want to study failure severity, whose empirical relational system is defined by the set of failures F and the relationship $more\_severe\_than$. Suppose we have a scale *m*, with values 1, 2, and 3. In this case, we can obtain a new measure *m'* by simply mapping these three values into any three values, provided that an order is defined among them and that the measure preserves that order. For instance, we can use the numeric triples 2, 4, 6, or 6, 15, 91, to obtain a

new scale. The set of admissible transformations in this case is broader than the set of admissible transformations for the distance between two points, since we are not forced to using only proportional transformations to obtain new scales. We could even use the triples a, b, c, or e, r, w, with the usual alphabetical ordering, or the three values low, medium, high, with the obvious ordering among them.

A broader set of admissible transformations means a broader choice of measures, but this comes with a price. All that our knowledge on failure severity allows us to tell is whether a failure is more severe than another, but nothing about the relative magnitude of severity. Therefore, it does not make sense to say that a failure is twice as severe as another failure, since the truth value of this statement depends on the specific scale used. For instance, one could say that medium severity failures are twice as severe as low severity ones according to the scale with values 1, 2, 3, or even the scale with values 2, 4, 6, but this would no longer be true according to the scale with values 6, 15, 91, let alone the scales with values a, b, c, or e, r, w. Instead, it makes sense to say that the distance between two points is twice as much as the distance between two other points. This statement does not change its truth value regardless of the scale chosen. This is a meaningful statement, according to the following definition.

Meaningful Statement. A statement about the values of a scale is meaningful if its truth value does not change if the scale is transformed according to any of its admissible transformations.

As an additional price to pay, we cannot use the values of the measure for failure severity as we would use the values we obtain for the distance between two points. For instance, we can sum the values we compute for distance measures, but it does not make sense to sum the values we obtain for failure severity.

5.5. Scale Types

The set of admissible transformations, the set of meaningful statements, and the set of operations that can be used on the values are related to each other. Specifically, the set of admissible transformation defines the *measurement level* of a scale. In general, the narrower the set of admissible transformations, the smaller the number of scales, and the more informative the scale we choose. In Measurement Theory, five measurement levels are usually used. Here, we describe them, from the least to the most informative one. They are summarized in Table 1 along with their properties.

5.5.1. Nominal Scale.

A scale is a nominal one if it divides the set of entities into categories, with no particular ordering among them. For instance, the programming language used for a program is a nominal measure, since it allows programs to be classified into different categories, and no ordering among them is available. The set of admissible transformations is the set of all one-to-one transformations, since the specific values of the measures do not convey any particular information, other than the fact that they are

different, so they can be used as labels for the different classes. For instance, a transformation that changed all the names of the programming languages would be acceptable, as long as different names are transformed into different names. We could have also used numbers to identify the programming languages, as long as different programming languages are encoded with different numbers. It is obvious that we cannot carry out any arithmetic operations on these numbers, since it makes no sense to sum or multiply them, and it does not make sense to order them, since there is no ordering of programming languages.

Nominal scales are the least informative ones, but they may well provide important information. For instance, a scale for the gender of people is a nominal one. However, scientific research has shown that gender may be related to diseases (e.g., the people of one gender may be more likely to have a disease than the people of the other gender) or to immunity to diseases. Therefore, nominal measures do provide important pieces of information. For instance, the programming language used is fundamental in interpreting models built on the number of lines of code.

5.5.2. Ordinal Scale.

A scale is an ordinal one if it divides the set of entities into categories that are ordered. For instance, an ordinal scale is the one that divides programs into small, medium, and large ones. The difference with the nominal scale is therefore that we have an additional piece of information, i.e., we know how to order the values of the measure. The set of admissible transformations is the set of strictly monotonic functions, i.e., those functions that preserve the ordering among the values of the measure, since we do not want to lose this piece of information. These functions are a subset of the one-to-one transformations, which are admissible for nominal scales. For instance, we can use the values a, b, and c with the usual alphabetical ordering instead of small, medium, and large. We might as well use the values 1, 2, and 3, with the usual, trivial ordering of integers. However, these values must then be used with caution. Unlike in the nominal case, we can compare these values, but we still cannot use them for arithmetic operations. For instance it would make no sense to sum the values 1 and 1 to obtain 2 and claim that 2 is the value of the size of a program segment obtained by putting together the two program segments whose values are 1 and 1. (Just imagine summing 3 and 3: the result would not even be defined.)

5.5.3. Interval Scale.

In an interval scale, the distance between two values is the basis for meaningful statements. For instance, it makes sense to say that the distance in time between the start of a software project and its completion is three times as much as the distance between the start and the end of the implementation phase. This statement makes sense regardless of the origin of time we adopt (e.g., foundation of Rome, first Olympic games) and the unit used (e.g., seconds, hours). The set of admissible transformations is a subset of the monotonic functions that are admissible for ordinal scales, the set of

functions of the kind $m' = am + b$, where b is any number (i.e., we can change the origin of the measure) and $a > 0$ (i.e., we can change the unit of the measure). Thus, not only can we order these distances, but we can establish truly numerical relations on them that are meaningful. What we cannot do is establish numerical statements of the same kind on the values of the measures. For instance, it does not make sense to say that the time at which a software project ended was twice as much as the time the project started. However, it makes sense to subtract values of an interval measure.

5.5.4. Ratio Scale.

A ratio scale allows the definition of meaningful statements of the kind ``twice as much'' on the single values of the measure. For instance one can say that a program segment is twice as long as another program segment. In this case, we can no longer choose the reference origin arbitrarily, as in the interval scale case. The reference origin is fixed, and its value is 0. On the other hand, we may change the unit without changing the meaningfulness of statements. For instance, when measuring the volume of objects, we can use liters, centiliters, gallons, etc. The truth value of a statement that says that the volume of an object is twice as much as the volume of another object does not change depending on the unit adopted. The set of admissible transformations is therefore a subset of the set of admissible transformations for the interval case, as the origin is now fixed. The set of admissible transformations is of the kind $m' = am$, where $a > 0$, since we can only change the unit of the measure. It makes sense to add and subtract ratio measures, and to multiply and divide them by constants.

5.5.5. Absolute Scale.

Absolute scales are the most informative ones, but they are also seldom used in practice. For instance, consider the attribute ``number of lines of code'' of a program segment. (This attribute is *not* the same as size, since we may want to measure size through different measures.) There is only one possible measure for this attribute, i.e., LOC, so there is only one admissible transformation, i.e., the identity transformation. Therefore, the set of admissible transformation, $m' = m$, is a subset of the admissible transformations for ratio measures.

5.5.6. Scale Types and Statistics

The above classification of scales has a very important impact on their practical use, in particular on the statistical techniques and indices that can be used. For instance, as an indicator of ``central tendency'' of a distribution of values, we can use the following different statistics, depending on the measurement level of the scale. For nominal scales, we can use the mode, i.e., the most frequent value of the distribution. (Several modes may exist.) For ordinal scales, we can use the median, i.e., the value such that not more than 50% of the values of the distribution are less than the median and not more than 50% of the values of the distribution are greater than the median. In addition, we can still use the mode. (There may be one or two—adjacent—medians.)

For interval scales, in addition to the median and the mode, we can use the arithmetic mean. For ratio and absolute scales, in addition to the arithmetic mean, the median, and the mode, we can use the geometric mean. Therefore, the higher the level of measurement of a scale, the richer the set of indicators of central tendency. More importantly, the higher the level of measurement, the more powerful the statistical techniques that can be used, for instance, for ascertaining the presence of statistical relationships between measures [7]. In particular, parametric statistical techniques may be used only with interval, ratio, and absolute scales, while only nonparametric statistical techniques may be used with nominal and ordinal scales. Without going into the details of these two kinds of statistical techniques, we can say that parametric statistical techniques may use additional information than nonparametric ones, specifically information about the distance between values that is available in interval, ratio, and absolute scales, but that cannot be derived in ordinal or nominal scales. Thus, fewer data are needed to reach a statistically-based conclusion with parametric techniques than nonparametric ones.

Table 1. Scale types and their properties.

Scale Type	Admissible Transformations	Examples	Indicators of Central Tendency
Nominal	Bijections	Name of programming language (attribute: ``programming language'')	Mode
Ordinal	Monotonically increasing	A ranking of failures (as a measure of failure severity)	Mode + Median
Interval	Positive linear	Beginning date, End date of activities (as measures of time)	Mode + Median + Arithmetic Mean
Ratio	Proportionality	LOC (as a measure for program size)	Mode + Median + Arithmetic Mean + Geometric Mean
Absolute	Identity	LOC (as a measure of the attribute ``number of lines of code'')	Mode + Median + Arithmetic Mean + Geometric Mean

5.6. Objective vs. subjective measures.

A distinction is sometimes made between ``objective'' and ``subjective'' measures. The distinction is based on the way measures are defined and collected. Objective measures are defined in a totally unambiguous way and may be collected through automated tools, while subjective measures may leave some room for interpretation and may require human intervention. For instance, ranking a failure as catastrophic, severe, non-critical, or cosmetic may be done based on human judgment. As a consequence, subjective measures are believed to be of lower quality than objective ones. However, there are a number of cases in which objective measures cannot be collected, so subjective measures are the only way to collect pieces of information that may be important and useful. Efforts must be made to keep the discrepancies among the values

that are provided by different people as small as possible. Guidelines can be provided for limiting the amount of variability in the values that different people can give to a measure. As an example, the definition of values for an ordinal measure should be accompanied by an explanation of what those values really mean. So, if we have a size measure with values, small, medium, and large, we need to explain what we mean by each of the three values. Otherwise, different people with different intuitions may provide totally different values when ranking a program according to that measure.

6. Property-based Approaches

Representational Measurement Theory is not the only way the properties of the measures for software attributes have been modeled. In recent years, a few studies have proposed to describe the characteristics of the measures for software attributes via mathematical properties, in the same way as other concepts have been described in the past (e.g., distance). Based on an abstract description of a software artifact, each attribute is associated with a set of properties that its measure should satisfy. A few sets of properties have been defined [8, 9], which address single software attributes, e.g., complexity, or general properties for software measures [10]. Here, we summarize the proposal of [11] because it addresses several different software product attributes. The proposal is based on a graph-theoretic model of a software artifact, which is seen as a set of elements linked by relationships. The idea is to characterize the properties for the measures of a given software attribute via a set of mathematical properties, based on this graph-theoretic model. We describe the basic concepts of the approach and the sets of properties for the measures of four internal software attributes of interest.

System, Module, and Modular System. A software artifact is modeled by a graph $S = \langle E, R \rangle$, called system, where the elements of a software artifact are modeled by the nodes, and the relationships by the edges. The subgraph that represents the elements and relationships of a portion of the artifact is called a module. So, $m = \langle E_m, R_m \rangle$ is a module of S if and only if $E_m \subseteq E$, $R_m \subseteq E_m \times E_m$, and $R_m \subseteq R$. A module is connected to the rest of the system by external relationships, whose set is defined as $OuterR(m) = \{\langle e_1, e_2 \rangle \mid (e_1 \in E_m \wedge e_2 \notin E_m) \vee (e_1 \notin E_m \wedge e_2 \in E_m)\}$. A modular system is one where all the elements of the system have been partitioned into different modules. Therefore, the modules of a modular system do not share elements, but there may be relationships across modules.

Figure 2 shows a modular system with three modules m_1, m_2, m_3.

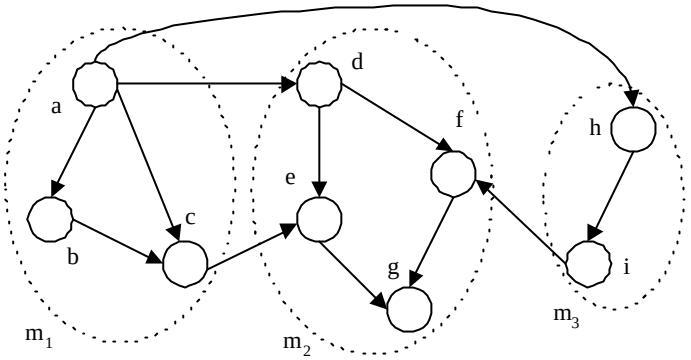

Fig. 2. A modular system.

Because of its generality, this graph-based model may be applied to several different kinds of software artifacts, including specifications, designs, code, etc. For instance, when modeling software code, elements may represent statements, relationships transfers of control from one statement to another, and modules functions.

A few additional definitions of operations and properties of modules have been defined, as follows ($m_1 = \langle E_{m1}, R_{m1} \rangle$, $m_2 = \langle E_{m2}, R_{m2} \rangle$, and $m = \langle E_m, R_m \rangle$).

- *Inclusion*. Module m_1 is included m_2 (notation $m_1 \subseteq m_2$) if and only if $E_{m1} \subseteq E_{m2}$ and $R_{m1} \subseteq R_{m2}$.
- *Union*. Module m is the union of modules m_1 and m_2 (notation $m = m_1 \cup m_2$) if and only if $E = E_{m1} \cup E_{m2}$ and $R = R_{m1} \cup R_{m2}$.
- *Intersection*. Module m is the intersection of modules m_1 and m_2 (notation $m = m_1 \cap m_2$) if and only if $E = E_{m1} \cap E_{m2}$ and $R = R_{m1} \cap R_{m2}$.
- *Empty module*. Module m (notation $m = \varnothing$) is empty if and only if $m = \langle \varnothing, \varnothing \rangle$. The empty module is the ``null'' element of this small algebra.
- *Disjoint modules*. Modules m_1 and m_2 are disjoint if and only if $m_1 \cap m_2 = \varnothing$.

We describe the properties for the internal attributes size, complexity, cohesion and coupling. Size and complexity may be defined for entire systems or modules of entire systems. Cohesion and coupling may be defined for entire modular systems or modules in a modular system. For simplicity, we only show the properties for the measures of cohesion and coupling of modules, and not of entire modular systems.

Size. The basic idea is that size depends on the elements of the system. A measure $Size(S)$ of the size of a system $S = \langle E, R \rangle$ is

- non-negative: $Size(S) \geq 0$

- null if E is empty: $E = \varnothing \Rightarrow Size(S) = 0$
- equal to the sum of the sizes of any two of its modules $m_1 = \langle E_{m1}, R_{m1} \rangle$, $m_2 = \langle E_{m2}, R_{m2} \rangle$ such that $\{E_{m1}, E_{m2}\}$ is a partition of E: $E = E_{m1} \cup E_{m2} \wedge E_{m1} \cap E_{m2} = \varnothing \Rightarrow Size(S) = Size(m_1) + Size(m_2)$

Complexity. Complexity depends on the relationships between elements. A measure $Complexity(S)$ of the complexity of a system $S =< E, R >$ is

- non-negative: $Complexity(S) \geq 0$
- null if R is empty: $R = \varnothing \Rightarrow Complexity(S) = 0$
- not less than the sum of the complexities of any two of its modules $m_1 = \langle E_{m1}, R_{m1} \rangle$, $m_2 = \langle E_{m2}, R_{m2} \rangle$ without common relationships: $R_{m1} \cap R_{m2} = \varnothing \Rightarrow Complexity(S) \geq Complexity(m_1) + Complexity(m_2)$
- equal to the sum of the complexities of two disjoint modules: $m_1 \cap m_2 = \varnothing \Rightarrow Complexity(S) = Complexity(m_1) + Complexity(m_2)$

Cohesion. Cohesion is based on the internal relationships of modules. A measure $Cohesion(m)$ of the cohesion of a module $m = \langle E_m, R_m \rangle$ of a modular system is

- non-negative and with an upper bound: $0 \leq Cohesion(m) \leq Max$
- null if R_m is empty: $R_m = \varnothing \Rightarrow Cohesion(m) = 0$
- non-decreasing if relationships are added to the set of relationships of a module: given two modules $m_1 = \langle E_{m1}, R_{m1} \rangle$, $m_2 = \langle E_{m2}, R_{m2} \rangle$, we have $R_1 \subseteq R_2 \Rightarrow Cohesion(m_1) \leq Cohesion(m_2)$
- non-increasing if two modules that are not linked by any relationships are grouped in a single module: $OuterR(m_1) \cap OuterR(m_2) = \varnothing \Rightarrow Cohesion(m_1 \cup m_2) \leq Cohesion(m_1) + Cohesion(m_2)$

Coupling. Coupling depends on the relationships that link a module to the rest of the system. A measure $Coupling(m)$ of the coupling of a module $m = \langle E_m, R_m \rangle$ of a modular system is

- non-negative: $Coupling(m) \geq 0$
- null if $OuterR(m)$ is empty: $OuterR(m) = \varnothing \Rightarrow Coupling(m) = 0$
- non-decreasing if relationships are added to the set of external relationships of a module: $OuterR(m_1) \subseteq OuterR(m_2) \Rightarrow Coupling(m_1) \leq Coupling(m_2)$

- not less than the sum of the couplings of any two of its modules $m_1 = \langle E_{m1}, R_{m1} \rangle$, $m_2 = \langle E_{m2}, R_{m2} \rangle$: $Coupling(m_1 \cup m_2) \leq Coupling(m_1) + Coupling(m_2)$
- equal to the sum of the couplings of two modules that are not linked by any relationships: $OuterR(m_1) \cap OuterR(m_2) = \varnothing \Rightarrow Coupling(m_1 \cup m_2) = Coupling(m_1) + Coupling(m_2)$

Sets of properties such as the one described above may be used to

- model intuition about the properties that measures for an attribute should possess
- show similarities and differences among the measures for different attributes
- check whether a given measure is consistent with intuition: if a measure satisfies the set of properties for an attribute does not imply that that measure is valid for the attribute it purports to measure, but as supporting evidence. On the other hand, if a measure does not satisfy the set of properties for the attribute it purports to measure, then it is not a measure for that attribute.

7. Measures for Internal Product Attributes

Many software measures have been defined over the last few years, mainly for programs. There are two reasons for this. Programs are the main products of software development, so they are always built during software development. The other artifacts may not always exist or they may not be explicitly and completely recorded. The second reason is that software code is a totally formal artifact, so it can be automatically processed to extract measures. Instead, the other software artifacts (e.g., requirements, specifications, etc.) are seldom fully formal, so they cannot usually be processed as easily in an automated way. In addition, human intervention may be required, so the measures obtained are more subjective than those for software code. Nevertheless, measures for artifacts other than software code exist, since measurement should be used in all phases of the software process, especially in the early ones, which are believed to have the biggest influence on the entire software process.

In what follows, we concisely describe a number of measures that have been defined for different software artifacts to measure a number of software attributes. Before proceeding, it is necessary to say that these measure have been seldom defined according to the concepts of Measurement Theory or the property-based approaches. This makes their adherence to intuition questionable in some cases.

7.1. Requirements

Measures for the early development phases would be the most useful ones, since the early phases and the artifacts they produce are believed to have the largest impact on the entire software development process and the final product. However, few

measures have been defined for the artifacts produced during the early phases. One of the reasons is that these artifacts are hardly ever formal in nature, and, therefore, it is hardly ever possible to analyze them in an automated fashion. Nevertheless, due to the need for early measures, measures for requirements have been defined.

7.1.1. Requirements Count

A widely used measure is requirements count, which is intended to measure the size of a software application. As such, it can be used to come up with at least a rough estimate of the size of the final product and, more importantly, time and effort required for developing the software product. This latter estimation may be carried out through mathematical estimation models (e.g., based on the average productivity in previous projects, interpreted, for instance, as the average time required for coding a requirement), or based on personal experience. The variations in the requirements count over time has also been used to quantify the volatility of requirements, which is a frequent phenomenon in industrial software development. Good definition and practical application of requirements count require that all requirements be of the ``same size,'' i.e., there shouldn't be requirements with intuitively different sizes, or, alternatively, requirements with intuitively different sizes should be given different ``weights.''

7.1.2. Function Points

Another measure has emerged over the years and is commonly used: Function Points. The aim of Function Points (FP) [12] is to measure software functionality. On an intuitive level, the more functionality in a program, the higher the number of Function Points. The attribute ``functionality,'' however, is quite elusive, so Function Points have been often taken as an *operational* definition of a measure for functionality, without much formal consideration on the intrinsic characteristics of the functionality attribute.

Function Points abstract from the specific languages used to specify, design, and implement a software system, so they are not a measure of the size of the final product. Due to the informal nature of requirements, Function Points are a subjective measure. Function Points are now used for several purposes, including contracts and pricing.

The computation of FP is carried out in two steps. First, Unadjusted Function Points (UFP) are computed. Second, a correction factor, Technical Correction Factor (TCF), is computed. The value of FP is the product $FP = UFP \cdot TCF$.

Unadjusted Function Points are computed based on element types belonging to the five categories below, which are used to summarize the input/output behavior of the software system and the internally handled data:

- external input types: each unique (different format or different processing logic) user data or control input type that enters the external boundary of the application and adds or changes data in a logical internal file type (include transactions from other applications);

- external output types: each unique user data or control output type that leaves the external boundary of the application (include reports and messages to the user or to other applications);
- logical internal file types: each major logical (from the viewpoint of the user) group of user data or control information that is generated, used, maintained by the application;
- external interface file types: files (major logical group of user data or control information) passed or shared between applications (outgoing external interface file types are also counted as logical internal file types);
- external inquiry types: each unique input/output combination, where an input causes and generates an immediate output (the input data is entered only to direct the search and no update of logical internal file type occurs).

Each element type of the software system (e.g., each different external inquiry type) is ranked on a three-valued complexity scale with values low, medium, high. A weight $W_{i,j}$ is associated with each element type i and complexity value j. Table 2 shows the values, which were determined by ``trial and error.'' Each element type therefore contributes to UFP according to its weight, so the value of UFP is obtained by summing all the products given by the number of elements types of kind i and complexity j times $W_{i,j}$

$$UFP = \sum_{i \in 1..5} \sum_{j \in 1..3} \#ElementTypes_{i,j} W_{i,j}$$

Table 2. Weights for the computation of Unadjusted Function Points.

Complexity / Function Types	Low	Average	High
External inputs	3	4	6
External outputs	4	5	7
Internal logical files	7	10	15
External interface files	5	7	10
External inquiries	3	4	6

The value of UFP is multiplied by a correction factor called Technical Complexity Factor (TCF), which accounts for 14 different General System Characteristics GSC_k of the software system: data recovery and back-up, data communication, distributed processing, performance issues, heavily used configuration, advanced data entry and lookup, online data entry, online update of master files, complex functionality, internal processing complexity, reusability, installation ease, multiple sites, modifiability. These characteristics are ranked according to their degree of influence $DegGSC_k$ from

``None'' to ``Essential'' and are accordingly associated with a value in the 0 – 5 range. The value of TCF is obtained as follows:

$$TCF = 0.65 + 0.01 \sum_{k \in 1..14} DegGSC_k$$

The idea is that the impact of each GSC_k on FP has a 5% range, the sum of the $DegGSC_k$'s is multiplied by 0.01 (i.e., 1%). In addition, if the influence of all GSC_k's is rated as ``Essential'' the value of $0.01 \sum_{k \in 1..14} DegGSC_k$ is 0.7. At the other extreme, if all the influence of all $GSC_k$'s is rated as ``None'' the value of $0.01 \sum_{k \in 1..14} DegGSC_k$ is 0. Therefore, there is a 70% oscillation interval, which explains the number 0.65 in the formula for TCF, since TCF varies in the $[0.65, 1.35]$ range.

Function Points are therefore computed based on subjective considerations. The extent of the discrepancies in the counting of Function Points by different people has been studied in a few works (e.g., [13]). At any rate, the International Function Points Users Group (IFPUG), an international organization with local chapters, periodically provides and refines existing guidelines to reduce the extent of subjectivity.

Several variants of Function Points have been defined. Among them, Feature Points [14] introduce Algorithmic Complexity as an additional element type and modify some values of the original complexity weights. Mark II Function Points [15] simplify the set of element types. A system is viewed as a set of ``transaction types,'' composed of input, processing, and output, used to compute the Unadjusted Function Points are. Then, Mark II Function Points are computed by multiplying the Unadjusted Function Points by a correction factor which takes into account 19 General System Characteristics.

The original justifications of Function Points were founded on Halstead's Software Science (see Section 7.4.2). Over the years, Function Points have been used as a measure of several attributes [16], including, size, productivity, complexity, functionality, user deliverables, overall behavior, or as a dimensionless number. Despite these theoretical problems, Function Points are widely spread and used as a *de facto* standard.

7.2. Specifications

Few measures exist for the attributes of software specifications. However, it would be important to have an early quantitative evaluation of the attributes of a software specification, since the early phases and artifacts are commonly believed to be the most influential ones during software development. The lack of measures for specifications is mainly due to the fact that specifications are usually written in plain text, so it is difficult to build ``objective'' measures that can be computed automatically. Measures

have been defined for formal or semi-formal specifications, since the measures can be defined in an ``objective'' way (i.e., no subjective assessments are required) and computations can be automated. As an example, a preliminary study was carried out on software specifications written in TRIO+ [17], a formal object-oriented specification language. Another application of software measurement to specifications is in [18].

7.3. Designs

A few measures have been defined for high-level and low-level design.

7.3.1. High-level Design Measures

In [19], measures have been defined for cohesion and coupling of the high-level design of an object-based system, which differs from a full-fledged object oriented system because objects are instances of types that are exported by modules, i.e., there is no real syntactic concept of class, and inheritance is not allowed.

At the high-level design stage, only the interfaces of modules of the system (functions types, variables, and constants defined in the module interfaces) are known.

The measures are based on two kinds of interactions, which may relate data to data or data to functions. There is an interaction between two data (DD-interaction) if one appears in the definition of the other (e.g., a type appears in the definition of another type, variable, constant, or function parameter). There is an interaction between a piece of data and a function as a whole (DF-interaction) if the piece of data appears in the definition of the function (e.g., a type appears as the type returned by the function or as the type of a parameter). The notions of interaction are also transitive, so, for instance, data A and C also interact if A interacts with a third piece of data B that interacts with C. Some interactions contribute to cohesion, other to coupling. The interactions between data and functions of a module and the interactions between data of a module (excluding the parameters of functions) are considered cohesive. The interactions between data of a module and data of other modules are believed to contribute to coupling. More specifically, some interactions contribute to import coupling of a module (those in which data from some other module appears in the data definitions of a module) and others to export coupling (those in which data from the module appear in the data definitions of some other module).

Among the measures defined for cohesion, the Ratio of Cohesive Interactions is the ratio of cohesive interactions existing in a module to the maximum number possible of cohesive interactions for that module, based on its data. Among the coupling measures, Import Coupling is the number of interactions that link data from other modules to data in a module. These measures are consistent with the properties shown in Section 6 for cohesion and coupling measures. In addition, it was shown that they could be used as a part of models for fault-proneness of the final software.

Among recent developments, measures have been proposed for object-oriented systems [20, 21], some of which may be applied to software designs and software code. The measures defined in [20] are explained in more detail in Section 7.4.4.

7.3.2. Low-level design measures

Based on the modularization of high-level design, low level design takes care of the designing the parts of the single modules. Information Flow Complexity [22] is probably the best-known measure for low-level design. The measure is based on the communication flows to and from a function. The input parameters and the global data structures from which the function retrieves information are called the fan-in. The output parameters and the global data structures that the function updates are called the fan-out. Information Flow Complexity is defined as follows:

$$InformationFlowComplexity = length \cdot (fanIn \cdot fanOut)^2$$

where *length* is a suitable size measure, and *fanIn* and *fanOut* are the number of parameters and data of the fan-in and fan-out, respectively.

7.4. Code

A very large number of measures have been defined for software code. Here, we report on some of the best-known and widely used ones.

7.4.1. Lines of Code

The number of Lines of Code (abbreviated as LOC) is the oldest and most widely used measure of size. It is commonly used to give an indication about the size of a program and in several models for the prediction of effort, fault-proneness, etc. The success of LOC is due to the fact that it is easy to understand and to measure. A few variations exist, though. For instance, one needs to decide whether to count only executable lines, as was done when LOC was first introduced, or declaration lines as well. Also, one needs to decide whether comment lines and blank lines should be counted. However, in the vast majority of cases, there is a very high linear correlation between the values of LOC obtained with or without comment and blank lines. Therefore, the predictive power is not really affected by these decisions. Nevertheless, it is sensible, in a specified development environment, to define a consistent mechanism for counting LOC, so the measures collected are comparable.

7.4.2. Halstead's Software Science

Halstead's Software Science [23] was an attempt to build a comprehensive theory for defining measures for several attributes of software code. The idea was to identify a set of basic elements of programs and measure them to build measures for a number of attributes of software code. Software Science argues that a program is composed of two basic types of elements, i.e., operators and operands. Operands are variables and constants. Operators are symbols or combinations of symbols that affect the values of operands. The basic measures computed on a program, on top of which Software Science was built, are:

- η_1 : the number of distinct operators that appear in the program
- η_2 : the number of distinct operands that appear in the program
- N_1 : the total number of occurrences of operators in the program
- N_2 : the total number of occurrences of operands in the program
- η_2^* : the number of conceptual input/output operands of in the program, i.e., the input and output parameters that would be needed if the program was represented as a function

Therefore, the main assumption of Software Science is that all measures for several attributes can be obtained based on these five measures. Through considerations deriving from a number of sources, including Information Theory and Thermodynamics, a number of measures are defined, as follows.

Program Length. A program is made only of operators and operands, so its length is given by the total number of occurrences of operators and operands, $N = N_1 + N_2$.

Length Estimator. Program length is known only upon program completion. However, it is useful to know or estimate the length of a program at the beginning of coding for prediction purposes, for instance to estimate the effort needed. Under the assumption that the number of distinct operators and operands of a program can be estimated early during coding, an estimator of length is derived as $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$.

Volume. The volume of a program is the number of bits needed to represent it, so it is another measure of size. It is computed as $V = N \log_2 (\eta_1 + \eta_2)$.

Potential Volume. This is the minimal number of bits required to represent a program, which would be attained if the function implemented by the program was already available. In that case, it is argued that the number of operators would be two (the name of the function and a mechanism to group the function's parameters), the number of operands would be η_2^*, and each operator and operand would appear only once, so the potential volume is $V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*)$.

Program Level. The level of the program (L) is an indicator of how close a program is to its potential volume, so $L = V / V^*$.

Estimator of Program Level. Since the value of η_2^* may be difficult to compute, the following estimator is introduced for *L*: $\hat{L} = 2\eta_2 / (\eta_1 N_2)$.

Effort. Effort is measured as the number of elementary discriminations (of the kind yes/no) that a programmer must make to write a program. It is argued that this number is $E = V^2 / V^* = V / L$. Since L may be difficult to obtain, due to the fact that it is computed based on η_2^*, $\hat{L}$ may be used in the formula.

Time. The time (in seconds) needed to write a program is given by the number of elementary discriminations divided by the number of elementary discriminations that a programmer makes every second, assumed to be 18, so $\hat{T} = E/18$.

The example in Figure 3 (in a Pascal-like language) shows how the computations are carried out.

	Operators	Occurrencies	Operands	Occurrencies

`begin`
`  max:=0;`
`  read(x);`
`  while x<>0 do`
`  begin`
`  if x>max`
`  then max:=x;`
`  read(x)`
`  end;`
`  write(max);`
`end;`
	`begin … end`	2	`max`	4

	`:=`	2	`0`	2
	`;`	5	`x`	5
	`read`	2		
	`(…)`	3		
	`while … do`	1		
	`<>`	1		
	`if … then`	1		
	`>`	1		
	`write`	1		
$\eta_2^* = 2$ (`x` and `max`)	$\eta_1 = 10$	$N_1 = 19$	$\eta_2 = 3$	$N_2 = 11$

$N = 30$, $\hat{N} = 38$, $V = 112$, $V^* = 11.61$, $L = 0.1037$, $\hat{L} = 0.0545$, $E = 1568$, $\hat{T} = 87.1$

Fig. 3. Example of computations for Halstead's Software Science.

Halstead's Software Science's theoretical foundations and derivations of measures are somewhat shaky, and it is fair to say that not all of the above measures have been widely used in practice. However, due to their popularity and the availability of automated tools for computing them, some of the above measures are being used. As an example, a value $V \leq 4000$ would be considered desirable for maintainability purposes in some environments.

7.4.3. Cyclomatic Complexity

The Cyclomatic Complexity [24] of a program is a measure for the complexity of the control flow graph of the program. The basic assumption is that the higher the number of paths in a program, the higher its control flow complexity. Since the number of paths in a control flow graph is infinite if the program has at least one loop, Cyclomatic Complexity only counts the so-called base paths, i.e., those paths from the start point to the end point of a program whose linear combinations provide all the paths in the program. Based on graph theory results, the number of base paths in a program is computed as $v(G) = e - n + 2$, where e and n are the number of edges and nodes in

the control flow graph, respectively. If a program has one main routine and $(p-1)$ subroutines, the value of $v(G)$ is computed as $v(G) = e - n + 2p$.

It is not necessary to build the control flow graph of a program to compute its Cyclomatic Complexity. Mills' theorem shows that $v(G) = d + p$, where d is the number of decision points in the program including its subroutines (an n-way decision point is counted n-1).

Figure 4 contains the control flow graph for the software code in Figure 3, for which $v(G) = 7 - 6 + 2 = 3$.

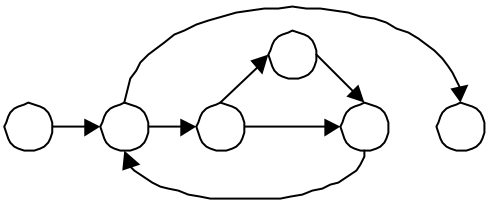

Fig. 4. Control flow graph for the program of Figure 3.

``Rules of thumb'' are sometimes used to curb the control flow complexity of software modules. In different application environments, upper limits ranging between 7 and 15 have been suggested.

7.4.4. Object-oriented measures

Following the success of Object-Oriented techniques, several measures have been proposed. The measures that have been most widely discussed and accepted are those defined in [20], which we now concisely describe. Some of these measures can also be used on object-oriented designs. We also provide a rough idea of the median values for them, based on the systems studied in [20, 25].

- *Weighted Methods per Class (WMC)*. It is the sum of the complexities of the methods of a class. Any complexity measure can be taken as the one to be used for the single methods, i.e., it is not fixed in the definition of the measure. By assuming each method has a unit weight, median values range between 5 and 10.
- *Number Of Children of a class (NOC)*, computed as the number of classes that directly inherit from a class in the inheritance hierarchy. Median values are around 0.
- *Depth of Inheritance Tree (DIT)*, which is the length of the longest path starting from the root of the inheritance hierarchy to the terminal classes. Median values range between 0 and 3.
- *Coupling Between Object classes (CBO)*, which is the number of classes to which a class is coupled, i.e., those classes that use that class or are used by that class. Median values may range around 5.

- *Response For a Class (RFC)*, defined as the number of methods of a class that can be executed when a class receives a message. Median values range between 15 and 20.
- *Lack of COhesion in Methods (LCOM)*, computed as the difference between the number of pairs of methods that do not share class variables (e.g., member data) and the number of pairs of methods that share class variables, if this difference is non-negative, otherwise LCOM is 0. Median values range between 0 and 1.

Although their theoretical validation is not fully convincing, the above measures (except LCOM) have been found generally useful in building models for various attributes. Many other measures have been defined. A comprehensive set is proposed in [21]. Surveys can be found in [26, 27].

7.4.5. Functional Cohesion

Three measures for functional cohesion are introduced in [28], based on the *data tokens* (i.e., the occurrences of a definition or use of a variable or a constant) that appear in a procedure, function, or main program. Each data token is on at least one *data slice*, i.e., a sequence of data tokens belonging to those statements that can influence the statement in which the data token appears, or can be influenced by that statement. Those data tokens that belong to more than one data slice are called *glue tokens*, and those that belong to all data slices are called *super-glue tokens*. Given a procedure, function, or main program p, the following measures SFC(p) (Strong Functional Cohesion), WFC(p) (Weak Functional Cohesion), and A(p) (adhesiveness) are introduced for all the data slices that refer to the results of the computations

$$SFC(p) = \frac{\#SuperGlueTokens}{\#AllTokens} \qquad WFC(p) = \frac{\#GlueTokens}{\#AllTokens}$$

$$A(p) = \frac{\sum_{GT \in GlueTokens} \#SlicesContainingGlueTokenGT}{\#AllTokens \#DataSlices}$$

It can be shown that $SFC(p) \le A(p) \le WFC(p)$, so the measures for cohesion define a range with lower bound $SFC(p)$ and upper bound $WFC(p)$ and ``central value'' $A(p)$. As an example, consider the function in Figure 5, which computes the minimum and the maximum values of a series of n integer numbers (with $n \ge 1$). The procedure computes two outputs (min and max), so we are interested in the data slices for those two outputs. For each data token in the procedure, the table in Figure 5 shows the occurrence number and whether it belongs to the min or the max data slice (YES or NO in the corresponding column). There are two data slices, so all glue tokens are also super-glue ones and the values for the three cohesion measures coincide, i.e., $SFC(p) = A(p) = WFC(p) = 8/22 = 0.36$.

```
procedure minmax
          (n: integer;
           var min, max: integer);
  var i: integer;
      temp: integer;
begin
  read(temp);
  min:=temp;
  max:=temp;
  for i:=2 to n do
  begin
    read(temp);
    if temp<min
      then min:=temp;
    if temp>max
      then max:=temp
  end
end;
```

Data Token	Occurrence	min	max
n	1	YES	YES
min	1	YES	NO
max	1	NO	YES
i	1	YES	YES
temp	1	YES	YES
temp	2	YES	YES
min	2	YES	NO
temp	3	YES	NO
max	2	NO	YES
temp	4	NO	YES
i	2	YES	YES
2	1	YES	YES
n	2	YES	YES
temp	5	YES	YES
temp	6	YES	NO
min	3	YES	NO
min	4	YES	NO
temp	7	YES	NO
temp	8	NO	YES
max	3	NO	YES
max	4	NO	YES
temp	9	NO	YES

Fig. 5. Example of computations for cohesion measures.

7.5. Verification

Measures have been defined for the artifacts used during the verification phase. Among these, the best known ones are coverage measures, which are used during software testing to assess how thoroughly a program is exercised by a set of test data. Coverage measures provide the percentage of elements of interest that have been exercised during software testing. Below, we list a few coverage measures that can be defined based on a given program P and a given set of test data TD.

- *Statement coverage* (also called *C0* coverage), i.e., the percentage of statements of P that are executed by at least one test in TD.
- *Branch coverage* (also called *C1* coverage), i.e., the percentage of branches of P that are executed by at least one test in TD.
- *Base path coverage*, i.e., the percentage of base paths (see Section 7.4.3) of P that are executed by at least one test in TD. Path coverage measures cannot be

used effectively, since the number of paths of the control flow graph of a non-trivial program (i.e., one that contains at least one loop) is infinite. Therefore, base path coverage or other measures are used instead.

- *Definition-use path coverage*, i.e., the percentage of definition-use paths of P that are executed by at least one test in TD. A definition-use path is defined as the path between the statement in which a variable receives a value and the statement in which that variable's value is used.

8. Measures for External Product Attributes

Several external attributes have been studied and quantified through measures. External attributes are usually attributes of industrial interest, since they are related to the interaction between software entities and their environment. Examples of these attributes are reliability, maintainability, and portability. Despite their practical interest, these attributes are often the least well-understood ones. One of the reasons is that they have several facets, i.e., there are several different ways to interpret them.

Reliability is one of the external software attributes that have received the most attention, because it probably is one of the best-understood one, and the one for which it is easiest to use for mathematical concepts. Reliability as an attribute has been studied in several other fields, including computer hardware. Therefore, a vast literature was available when software reliability studies began and an accepted definition of reliability was available. For software engineering applications, reliability may be defined as the probability that a specified program behaves correctly in a specified environment in a specified period of time. As such, reliability depends on the environment in which the program is used. For instance, it is to be expected that a program's reliability is different during testing than it is during operational use. There is an important difference between software and other traditional fields in which reliability has been studied. Software does not wear out, as does hardware, for instance. Thus, software reliability studies have devised new reliability models that can be used in the software field. Most of these models [29] provide an estimate of reliability based on the time series of the failures, i.e., the the times at which failures have occurred.

Another important product attribute is the detectability of faults in a software artifact. It is an external attribute in that it depends on the environment (e.g., techniques) and the time span in which the software artifact is examined. Therefore, it is not to be confused with the presence of faults in a software artifact. Faults may be detected in all software artifacts, and the nature of the specific artifact at hand influences the type of methods used to uncover faults. For instance, testing may be used to uncover faults on formal artifacts, such as programs, but executable specifications as well. As another example, inspections may be used with most sorts of artifacts. The simplest measure of the detectability of faults is the number of faults uncovered in a software artifact. This number may provide an idea of the quality of the artifact. However, it is usually useful to identify different categories of faults, which can be for instance classified according to their criticality from the user's viewpoint or the

software developers' viewpoint. A problem may be the definition itself of a fault. For instance, suppose that a variable is used in several points of a program instead of another one. Does the program contain one fault or is every occurrence of the incorrect variable to be considered a single fault? Unambiguous guidelines should be provided and consistently applied to prevent these problems. A related attribute is the detectability of software failures in software products.

9. Measures for Process Attributes

A number of process attributes are well understood, because they are closer to intuition than product attributes. For instance, these attributes include development effort, cost, and time. Process attributes often have an industrial interest. Building measures for these attributes is not a problem, as long as some consistent criterion is used, so measures are reliable and comparable across projects or environments. Process attributes are also used for tracking software project progress and checking whether there are problems with its schedule and resources. A number of derived attributes are of industrial attributes. For instance, productivity is a popular attribute and may have different facets depending on the process phase in which it is applied. As an example, in the coding phase, one may interpret productivity in terms of the amount of product or functionality delivered per unit time; in the verification phase, productivity may be interpreted in terms of the failures caused per unit time. These derived attributes may pose a definition problem. For instance, measuring coding productivity as the number of lines of code produced in the unit time may unjustly reward long programs. A more general problem that involves many process attributes is prediction, i.e., managing to link their measures with measures for internal product and process attributes, especially those that are available in the early development phases. Examples on how measures for process attributes may be used are shown in Section 10.3.

Software process measurement is an important part of improvement and maturity assessment frameworks. An example of improvement framework is the QIP, as briefly mentioned in Section 2. Several maturity assessment framework exist, among which the Capability Maturity Model (CMM) [30] is probably the best known one. The CMM defines five maturity levels at which a software organization may be classified. At each level, the CMM defines a number of Key Problem Areas (KPAs), i.e., issues that a software organization needs to address to progress up to the next maturity level. Process measurement is one of the KPAs addressed at the CMM level 3 (Defined Level), so the software organization may reach the CMM level 4 (Managed Level). The reader may refer to [31] for further information on software process measurement.

10. Practical Application of Measurement

There are several important aspects in the practical application of measurement, all of which derive from the fact that measurement should be seen as an activity that provides value added to the development process.

10.1. Experimental Design

The measurement goals also determine the specific ``experimental design'' to be used. We have put ``experimental design'' in quotes, because that is the term used by the scientific literature, but application of software measurement does not necessarily entail carrying out a full-fledged scientific experiment, i.e., one where hypotheses are confirmed or disconfirmed. What we mean here is that the measures to be defined, the data to be collected, and the results that can be expected should be clearly specified, and that a suitable measurement process should be put in place. At any rate, it is not the goal of this paper to describe the various experimental designs that can be used in software measurement even when one wants to carry out a real scientific experiment. We simply want to highlight that it is important to identify an experimental design that is consistent with the defined goals and the expected results. This might seem obvious, but problems sometimes have arisen in the scientific literature and practical applications for a variety of reasons, e.g., the measures were not the right ones for the quantification of the attributes, the specific experimental design could not lead to the stated conclusions, other reasons could be found for the results obtained, hidden assumptions were present, etc.

10.2. Data Collection

Data collection is not a trivial part of software measurement, and may well take a large amount of time and effort. Therefore, data collection should be designed and planned carefully. Based on the measures that have been defined, data collection requires the identification of

- all the points of the process at which data can and should be collected
- the artifacts that need to be analyzed
- who is responsible for collecting data
- who should provide data
- the ways in which data can/should be collected (e.g., via questionnaires, interviews, automated tools, manual inspection)

Data collection should be as little invasive as possible, i.e., it should not perturb the software development process and distract developers from their primary goals. Therefore, all activities that can be automated should be automated indeed. This can be done by building automated analyzers for software code, to obtain data for the measures of internal software attributes.

More importantly, the collection of data that involves human intervention should be automated to the extent possible. For instance, electronic data forms should be used to collect project data from the software developers and managers, instead of other kinds of data collection. These forms should be

- complete but concise, i.e., ask for only for the information that is necessary to collect from developers and managers so that they should not use much time filling the forms out

- clear, i.e., the meaning of the information that is required should be straightforward to developers and managers, so they can provide accurate and precise answers in little time

Automated collection has also the positive effect of feeding the data directly into the database that contains the values, so they can be used for interpretation and analysis.

Data collection should be timely. Sometimes, data are reconstructed *a posteriori*. Though this procedure may provide useful pieces of information, there is a danger that these data are not reliable.

Not all the collected data can be used. Some data may be incomplete (e.g., some important pieces of information are missing), inconsistent (e.g., they conflict with each other), or incorrect (e.g., some values may be out of the admissible range). Before the data analysis, all these data should be removed from the database and lessons should be learned to prevent—to the extent possible—the reoccurrence of such problems.

10.3. Data Analysis

Data analysis can be used to carry out the so-called empirical validation of software measures, i.e., show that a measure is useful towards some parameter of industrial interest (as opposed to the so-called theoretical validation of measures as described in Sections 5 and 6, which entails providing sufficient evidence that a measure really quantifies the attribute it purports to measure). This is obtained through the building of models, based on a number of techniques.

It would be well beyond the scope of this article to provide a thorough examination of the techniques that can be used to analyze software measurement data, since a vast literature is available about data analysis. At any rate, we would like to point out that, beside well-known and well-established statistical techniques such as linear regression or principal component analysis [32], there are a number of other techniques that have been used for data analysis in software engineering measurement. However, it is necessary to point out a few problems of statistical analyses (e.g., see [33]). For instance, the existence of a statistical correlation between two variables does not show the existence of causal relation between them. In addition, given a large number of variables, there is a high likelihood that at least two of them are linked by a statistical correlation. Therefore, some care should be used in interpreting a statistical correlation relation between variables.

New statistical techniques have been borrowed from other disciplines. For instance, Logistic Regression [34] has been used originally in medicine. In its simplest form, Logistic Regression estimates the probability that an object belongs to either of two classes, based on the values of measures of attributes of the object. Based on a threshold value on the estimated probability, objects are classified as belonging to either class. For instance, Logistic Regression has been used in software measurement to predict whether a software module is faulty based on the values of measures collected on the module.

Machine-learning based techniques have been borrowed from artificial intelligence, e.g., classification trees [35]. Other machine-learning techniques, e.g., Optimized Set Reduction [36] have been defined in the context of software measurement. These techniques divide a set of objects into subsets in a stepwise fashion based on the values of measures for classification purposes. The goal is to obtain subsets that are homogeneous with respect to the values of a measure for some attribute of interest. For instance, these techniques can be used to predict which modules are faulty.

Models have been built that relate measures for internal product attributes and process attributes, on the one side, to measures for external product attributes or process attributes, on the other side. For instance, models have been defined that relate a number of measures such as LOC, v(G), etc., to software fault-proneness, cost, etc. (a review can be found in [1]). Among the best known ones, COnstructive COst MOdel (COCOMO) [37] defines a family of models that allow for the estimation of software cost based on LOC and a number of attributes belonging to four categories, i.e., product attributes, computer attributes, personnel attributes, and project attributes. The measures for these attributes are computed in a subjective way. COCOMO has undergone several changes over the years, with the new COCOMO 2 [38].

At any rate, the derivation and use of sophisticated models may not be justified in various application cases. For instance, suppose that software measurement is used in a software project for tracking purposes and the focus is on elapsed time, to check whether the project is being carried out according to the original schedule. A simple comparison between the original schedule and the actual one will be probably sufficient for identifying schedule slippage. Figure 6 shows the comparison between two simple GANTT diagrams of a project that is running behind its original schedule.

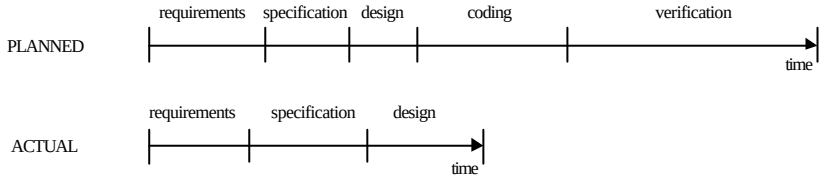

Fig. 6. Simple GANTT diagrams for a project.

As another example, Figure 7 shows the number of failures (divided according to their criticality level) found in a series of consecutive baselines (i.e., intermediate releases) through which a software product goes during coding and testing.

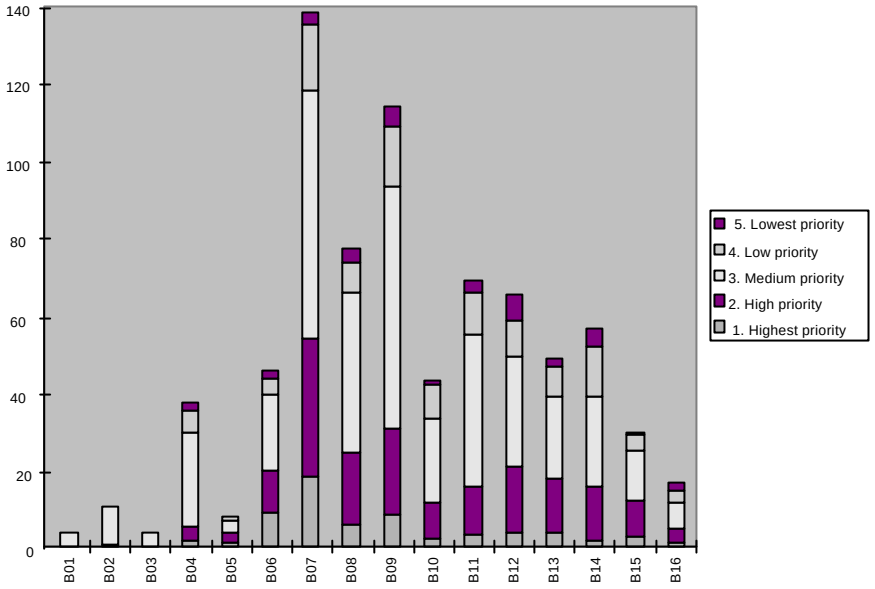

Fig. 7. Failures per priority per baseline.

The data in Figure 7 may be used during coding and verification to monitor, evaluate, and control the software process. For instance, Figure 7 shows that the number of failures decreases in the last baselines, which suggests that the number of faults in the software is decreasing too. The figure also shows that the number of highest priority failures is kept under control. More information on the topic may be found in [39, 40].

10.4. Interpretation and Use of Results

The final goal of measurement in industrial environments is not to validate measures or build models, but to make the results obtained available to software practitioners, who can use them in future projects. First, measurement results should be interpreted by the software developers and managers of the environment(s) in which the data have been collected. They are the ones that have the necessary knowledge to understand and explain the phenomena that have been studied through measurement. Researchers might provide tentative explanations and interpretations, but these can hardly be conclusive if they are not validated by software practitioners. For instance, only the knowledge of the software process and the specific project may explain why there are few failures before baseline 4 in the project whose data are shown in Figure 7 and there is a peak for baseline 7.

Second, it is imperative to understand how the measurement results are to be used. It should be clear from the start of a measurement program that measurement will not be used to judge people, but to help them. If this is not clear, the measurement program

is doomed to failure, because a measurement program should always be seen as a part of an improvement process, which entails providing better ways for software practitioners to carry out their duties. In addition, there is a danger that software practitioners will not cooperate in data collection or, if they do, provide reliable information. Thus, the measurement program will be doubly unsuccessful, in that it provides bad data and results that are not used to improve software development [39].

10.5. Standards

Standards for software quality have been defined. The best known of these is probably ISO/IEC 9126 [41], which defines software quality as composed of six external attributes of interest, namely, functionality, reliability, efficiency, usability, maintainability, and portability. In turn, each of these qualities is refined into sub-attributes. For instance, maintainability is refined into analyzability, modifiability, stability, and testability. The ``weights'' of the external attributes and sub-attributes may vary according to the product whose quality is under evaluation. The main problems with this standard and other quality models (e.g., see McCall's quality model [42]) are that it is not clear

- whether the set of attributes is complete
- whether an attribute should be such or it should be a sub-attribute, and whether a sub-attribute should be an attribute, instead
- what the definitions are for the attributes and their measures (as explained in Sections 5 and 6).

11. Future Developments

Future work on software measurement will encompass both theoretical and practical activities. On the theoretical side, studies are needed to better characterize the relevant attributes to be studied and the properties of their measures. On the application side, measurement needs to be introduced in traditional development environments and in new ones, such as web-based applications.

References

1. N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. (International Thomson Publishing, London, 1996).
2. IEEE Software, Special Issue on Measurement, **14** (1997).
3. V. R. Basili and D. Weiss, ``A Methodology for Collecting Valid Software Engineering Data,'' *IEEE Trans. Software Eng.* **10** (1984) 728-738.
4. V. R. Basili and H. D. Rombach, ``The Tame Project: Towards Improvement-Oriented Software Environments,'' *IEEE Trans. Software Eng.* **14** (1988) 758-773.
5. F. S. Roberts, *Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences* (Addison-Wesley, Reading, MA, 1979).
6. H. Zuse, *A framework of software measurement* (Walter de Gruyter, Berlin, 1998).

7. M. Kendall and A. Stuart, *The Advanced Theory of Statistics*, 4th ed. (Charles Griffin & Co., London, UK, 1977).
8. R. E. Prather, ``An Axiomatic Theory of Software Complexity Measure," *The Computer Journal* **27** (1984) 340-346.
9. E. J. Weyuker, ``Evaluating Software Complexity Measures," *IEEE Trans. Software Eng.* **14** (1988) 1357-1365.
10. J. Tian and M. V. Zelkowitz, ``A Formal Program Complexity Model and Its Application," *J. Syst. Software* **17** (1992) 253-266.
11. L. C. Briand, S. Morasca and V. R. Basili, ``Property-based software engineering measurement," *IEEE Trans. Software Eng.* **22** (1996) 68-86.
12. A. J. Albrecht and J. Gaffney, ``Software function, source lines of code and development effort prediction," *IEEE Trans. Software Eng.* **9** (1983) 639-648.
13. D. R. Jeffery, G. C. Low and M. Barnes, ``A comparison of function point counting techniques," *IEEE Trans. Software Eng.* **19** (1993) 529-532.
14. C. Jones, *Programmer Productivity* (McGraw-Hill, New York, 1986).
15. C. R. Symons, ``Function point analysis: difficulties and improvements," *IEEE Trans. Software Eng.* **14** (1988) 2-11.
16. A. Abran and P. N. Robillard, ``Function points: a study of their measurement processes and scale transformations," *J. Syst. and Software* **25** (1994) 171-184.
17. L. C. Briand and S. Morasca, ``Software measurement and formal methods: a case study centered on TRIO+ specifications," in *Proc. ICFEM'97*, Hiroshima, Japan (Nov. 1997) pp. 315-325.
18. K. Finney, K. Rennolls and A. Fedorec, ``Measuring the comprehensibility of Z specifications," *J. Syst. and Software* **42** (1998) 3-15.
19. L. C. Briand, S. Morasca and V. R. Basili, ``Defining and validating measures for object-based high-level design," *IEEE Trans. Software Eng.* **25** (1999) 722-741.
20. S. R. Chidamber and C. F. Kemerer, ``A metrics suite for object oriented design," *IEEE Trans. Software Eng.* **20** (1994) 476-498.
21. F. B. Abreu and R. Carapuca, ``Candidate metrics for object-oriented software within a taxonomy framework," *J. Syst. and Software* **23** (1994) 87-96.
22. S. Henry and D. Kafura, ``Software structure metrics based on information flow," *IEEE Trans. Software Eng.* **7** (1981) 510-518.
23. M. Halstead, *Elements of software science* (Elsevier, North Holland, New York, 1977).
24. T. J. McCabe, ``A Complexity Measure," *IEEE Trans. Software Eng.* **2** (1976) 308-320.
25. V. R. Basili, L. C. Briand and W. L. Melo, ``A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Software Eng.* **22** (1996) 751-761.
26. L. Briand, J. Daly and J. Wuest, ``A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering - An International Journal* (1998).
27. L. Briand, J. Daly and J. Wuest, ``A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Trans. Software Eng.* (1999).
28. J. Bieman and L. M. Ott, ``Measuring Functional Cohesion," *IEEE Trans. Software Eng.* **20** (1994) 644-657.
29. J. D. Musa, A. Iannino and K. Okumoto, *Software reliability: prediction and application* (McGraw-Hill, New York, 1987).

30. M. C. Paulk, C. V. Weber and B. Curtis, *The Capability Maturity Model: Guidelines for Improving the Software Process* (Addison-Wesley 1995).
31. IEEE Software, Special Issue on Measurement-Based Process Improvement, **11** (1994).
32. J. C. Munson and T. M Khoshgoftaar, ``The detection of fault-prone programs,'' *IEEE Trans. Software Eng.* **18** (1992) 423-433.
33. R. E. Courtney and D. A.Gustafson, ``Shotgun correlations in software measures,'' *Software Engineering Journal* (January 1993) 5-13
34. D. W. Hosmer and S. Lemeshow, *Applied Logistic Regression* (John Wiley & Sons, Inc., New York, 1989).
35. R. W. Selby and A. A. Porter, ``Learning from examples: generation and evaluation of decision trees for software resource analysis,'' *IEEE Trans. Software Eng.* **14** (1988) 1743-1757.
36. L. C. Briand, V. R. Basili and W. M. Thomas, ``A pattern recognition approach for software engineering data analysis,'' *IEEE Trans. Software Eng.* **18** (1992) 931-942.
37. B. W. Boehm, *Software engineering economics* (Prentice-Hall, Englewood Cliffs, 1981).
38. B. W. Boehm, B. Clark, E. Horowitz et al., Cost models for future life cycle processes: COCOMO 2.0,`` *Annals of software engineering* **1** (1995) 1-24.
39. R. B. Grady, Practical software metrics for project management and process improvement (Prentice-Hall, Englewood Cliffs, 1992).
40. W. Royce, Software Project Management : A Unified Framework (Addison-Wesley, 1998).
41. International Standards Organization, Information Technology – Software product evaluation – Quality characteristics and guidelines for their use, ISO/IEC9126, Geneva, Switzerland, 1991.
42. J. A. McCall, P. K. Richards and G. F. Walters, ``Factors in software quality,'' RADC TR-77-369, 1977 (Rome Air Development Center).