MEASUREMENT

INTRODUCTION

The objective of this article is to provide an overview of software measurement and how it has progressed over the past few years. To begin with, the terms *software measure* and *software measurement* will be defined as they will be used throughout the article.

- A *software measure* is a mapping from a set of objects in the software engineering world to a set of objects in the mathematical world. Objects in the software engineering world may be projects, products and processes. Objects in the mathematical world may be numbers or vectors of numbers. These mappings can be defined on different scales such as nominal, ordinal, interval, or ratio (Zuse, 1991).

A measure can be used to characterize some property of a (class of) software engineering object(s) quantitatively. For example, the measure *Lines-of-Code* characterizes the property *size* of software source code by associating a number (i.e., number of lines of code) with it. The measure *Number-of-Design-Faults* characterizes the property *error* *proneness* of the software design process by associating a number (i.e., number of design faults detected) with it. The measure *Number-of-Staff-Hours* characterizes the property *resources-consumed* of a software project by associating a number (i.e., number of staff-hours used) with it.

- *Software measurement* is a technique or method that applies software measures to a (class of) software engineering object(s) to achieve a predefined goal. Such goals of measurement vary along five characteristics (Basili, 1989; Basili and Rombach, (1988): what software engineering objects are being measured, why they are being measured, who is interested in these measurements, which of their properties are being measured, and in what environment they are being measured.
- *Object of measurement*. People measure different software engineering objects ranging from products to processes and entire projects. Example products are source code components, software designs, software requirements, and software test cases. Example processes are the architectural design process, the coding and unit test process, and the system test process.
- *Purpose of measurement*. People measure software engineering objects for different purposes. Examples include characterization, assessment, evaluation, prediction, and improvement.
- *Subject of measurement*. Different people or organizations are interested in measuring software engineering objects. Examples are the software designer, software tester, software manager, software quality ensurer, and the entire software development organization.
- *Measured property*. People may be interested in different properties of software engineering objects. Example properties include cost, adherence to schedule, reliability, maintainability, and correctness.
- *Context and environment measurement*. Software engineering objects are measured in different environments. Example characteristics of an environment may be the kinds of people involved, technology used, applications tackled, and resources available.

The following example goals are formulated according to the above five dimensions:

- To "characterize" the "cost" of "projects" from a "coporation's viewpoint," so that the planning of future projects (e.g., how much effort will a new project cost?) can be supported.
- To "characterize" "weaknesses and strengths" of the "current processes and products" from a "project manager's viewpoint," so that it is known what problems have to be expected (e.g., what types of errors are commonplace?).
- To "assess" the "effectiveness" of "candidate techniques" in dealing with certain problems from a "project planner's viewpoint" (e.g., does functional testing minimize certain fault classes?).
- To "characterize" "problems" with the "current process and products" and the "effectiveness" of "candidate techniques" in dealing with those problems from a "planner's viewpoint," so that the most effective techniques can be adopted (e.g., which technique will minimize current problems?).
- To "evaluate" the "quality" of the "process–product" from a "quality ensurance person's viewpoint" (e.g., what is the reliability of the product after delivery?).
- To "evaluate" the "functionality and user friendliness" of the "delivered product" from a "customer's viewpoint" (e.g., does the product provide the needed functionality and is it easy to use?).

Each of the above example goals requires a different set of measures, and the data collected according to these measures require different interpretations. Measurement must be sensitive to different goals.

HISTORY OF MEASUREMENT

In this section we will provide an outline of the historical steps that have lead us to the current status in the area of measurement. The discussion will be centered around the concept of goal of measurement and its dimensions as introduced in the last section.

During the sixties, seventies and early eighties software measurement was in its primitive stage. This is reflected by the fact that measurement goals were not explicit nor comprehensive. Implicit goals reflected the "production" view that there was a common set of goals shared by the entire software community. People disagreed over the usefulness of measures and measurements without realizing that they had different goals in mind. How else could it be possible that entire papers were devoted to discuss the usefulness of individual measures out of context. The lack of comprehensiveness is best demonstrated by the lack of coverage along our five goal dimensions.

Let us review some of those goals:

- The scope of measurement was mostly limited to entire projects and products at the code level. The importance of upstream products and processes for improvement were not fully recognized.
- The main purpose was to control project- and product-level software properties. The necessity of first understanding the cause-and-effect relationships between various project, process and product factors was underestimated.
- The dominant perspective was those of project managers who wanted to gain control over their projects. The need of all people involved in software developments for quantification was not clearly understood.
- The main property being measured were defects and cost at the project level, and volume–structure of the resulting code. Example project measures characterized product quality as perceived by the customer (e.g., customer trouble reports), schedule (e.g., calendar start and completion times for major project

phases), and cost (e.g., number of staff-months). Example product measures (typically confined to the code level) characterized size (e.g., lines of code) and complexity (e.g., number of control paths). The implicit assumption was that there exists a fixed number of properties of interest, rather than to be flexible and allow the user of measurement to identify any property of interest.

- Measurement was typically performed in constrained environments (e.g., individual projects within the same organization). The characteristics of these environments were not captured explicitly. This reflected the fact that people did not understand the dependency of models and measures on environment characteristics. There were enough indications of this misconception when people attempt to reuse models and measures naively across different environments without revalidation.

Hardly any measurement methodologies existed. Measurement was perceived as technology that could be applied as an add-on to software development projects. As a consequence of this approach:

- Measures were selected, interpreted, and used based on project managers' intuition. The underlying attitude was that there exists a standard set of development processes, models and measures. This production-style view simplified the selection of measures and their use. However, it did not do justice to the complexity of software products and its development processes.
- Data collection and especially validation were not emphasized (Basili and Weiss, 1984).
- The production-style view of software developments also gave the dangerous impression that measures and related results could be reused across project and even organizational boundaries without modification or revalidation.

The measurement activities automated were typically limited to libraries for storing trouble reports and tools for tracking resources such as computer time.

Practical applications of software measurement were limited to the collection of trouble reports (i.e., failures according to the IEEE terminology (1983). Analysis was hampered by the fact that little data were collected regarding the causes of failures (i.e., faults and errors). From that point of view, trouble statistics gave a snapshot of the defect quality of individual projects without the possibility of doing better in future projects.

Most researchers concentrated their efforts on developing new models and measures. Examples range from project resource (Basili, 1980) and defect (Belady and Lehman, 1976) to code size (Halstead, 1977) and complexity (McCabe, 1976), models and measures. Many studies conducted during that period did not describe the environment, goals, and experimental procedures to be reused by other researchers.

Currently the situation is characterized by a community consensus (Rombach and co-eds, 1993) that:

- The scope of measurement needs to include entire projects, products at all levels (i.e., from requirements to code and test) and processes. This requires better explicit models especially for upstream products and processes.
- The purposes of measurement are understanding, planning and control, and improvement, in this order.
- The persepctives have to reflect all roles people play in software projects: planners, managers, quality assurance, developers, etc.
- The properties being measured depend on the scope, the purpose and the perspective of interest. There is no fixed set of measures for all goals and environments.
- Empirically based models can be established for software development in specific environments.
- The impact of changing environments on those models is currently subject of studies.

As far as measurement technologies are concerned, significant progress has been made regarding

- Models for continuous improvement that provide the appropriate context for measurement: Quality Improvement Paradigm, Total Quality Management, SEI Capability Maturity Model.
- Models for goal oriented measurement.
- Better infrastructure for experiment design, data collection and validation, and analysis.

The future agenda in software measurement, as it looks today, includes

- Integration of measurement into automated software engineering environments.
- Integration of modeling and measurement technology for processes.
- Rigorous and appropriate infrastructures and methodologies for collection, validation and analysis of software engineering data.
- Support of industrial paradigms and software factories.
- Frameworks for experimentation and analysis in large industrial environments.
- Inclusion of quantitative and experimental approaches into teaching curricula, training programs and technology transfer approaches.

SOFTWARE MODELS AND MEASURES

It is important to have many kinds of models and measures. Models and measures are inseperable companions. Measures are intended to characterize some property of a software engineering object in quantitative terms. However, different models may exist for the same property. It is, therefore, impossible to judge the appropriateness of a chosen measure without understanding the underlying property model the measure is supposed to quantify. In the following sections the different types of measures and models needed are discussed and examples of project, product, and process models and measures are given.

Types of Measures

There is a distinction between objective and subjective measures, abstract and specific measures, complex and simple measures, product and process measures, and direct and indirect measures. There is a lot of information that cannot be measured objectively; it may be an estimate of the extent or degree in the application of some technique, a classification, or qualification of problem or experience, usually done on a nominal or ordinal scale. An example may be the subjective measure *team experience with some technology*. Such a measure can be defined in an operational way on a nominal scale as follows:

0. None.
1. Have read the manuals.
2. Have had a training course.
3. Have had experience in a laboratory setting.
4. Have used it on a project before.
5. Have used it on several projects before.

Defining each value between 0 and 5 in an operational sense has the advantage of guaranteeing consistent data collection. Without such operational definitions, subjective measures are usually limited, because it is in the eye of the beholder to judge a given team as having an experience level of either 0 or 5.

Most measures reported in the literature are abstract measures in the sense that they are derived from some abstract model. They need to be tailored to the specific characteristics of an environment before they can be applied. Those tailored measures are called specific. For example, the abstract measure of fan-in–fan-out needs to be tailored to specific languages such as Fortran or Ada before it can be applied practically.

Simple measures are those based on a single property; complex measures are those based on a vector of properties. For example, the project measure *number of faults* is a simple measure; the measure *number of faults per 1KLoC* is a complex measure because it is based on two properties: *errorproneness* and *size*.

The need for product, project, and process measures was noted earlier. Direct measures of a property are measures suited to define this property quantitatively. Indirect measures of a property are measures suited to predict the values of direct measures. Very often, direct measures (e.g., number of failures per unit of operation as a direct measure of reliability) can only be measured after delivery of the product. Indirect measures (e.g., number of failures per unit of testing or complexity of the product) enable one to explain or even predict the values of the direct measures early.

Types of Models

Measures are organized into models in two ways:

- Attribute Models: the model states that an external attribute of a software artifact is represented by one or more internal attributes direclty associated with measures; e.g., the size of a software module is represented by lines of code. Models of this kind are generally used to characterize a qualitative aspect in a quantitative way.
- Relationship Models: The model establishes a (generally functional) relationship among one attribute and a list of other ones; e.g., the effort needed to develop a code module is a function of the number of lines of code in the module. Models of this kind are used either to represent the relationship among quantative aspects or to describe a relationship among qualitative aspects in terms of their quantitative characterizations.

A well known example of relationship models is provided by the so called cost models such as COCOMO (Boehm, 1981). They are mostly automated and proprietary. The experience reported is mixed: they seem to work in certain environments, but not in others. It seems to be hard to tailor them to the characterics of different environments. Most of these models predict "project cost" based on the "anticipated size of a product" (e.g., Lines of Code), "the complexity of the product requirements" (e.g. function points), and other subjective measures of the environment such as "experience of personnel" or "technology level used for development". Typical resource allocation models describe the staffing profile and/or project schedule.

Most product models are still emphasizing software code. Static code models, reliability models, and other defect models exist. Static code models are primarily based on size and complexity. The most popular size and complexity measures are Halstead's (1977) software science measures and McCabe's (1976) cyclomatic complexity measure as well as variations thereof. In addition, data-based complexity measures such as information flow measures (S. Henry and Kafura, 1981) and data binding measures (Basili and Hutchens, 1983) have been used as alternatives to McCabe-style measures to characterize complexity. The experience is also mixed: they seem to work in certain environments but not in others.

Many reliability models have been published (Husa, 1980). These models are typically intended to assess the quality of a software product and require independent failures. For that reason , these models are clearly misapplied on data derived from testing techniques oriented toward defect detection. Nevertheless, there are many application of the latter kind reported in the literature. Other problems with reliability models include the evaluation of underlying model assumptions, the refinement of models for particular applications, the selection of approaches for using models, and the application of models to specific projects.

Other popular and useful defect models are based on relating error, fault, and failure profiles based on a variety of classification schemes (Basili and Rombach, 1987; Basili and Weiss, 1984). Such models provide a good basis for characterizing software projects and entire organizations from a defect quality perspective. They also enable cause-and-effect analyses as a first step toward improvement, either in terms of better corrective (i.e., defect detection and correction) or preventive (i.e., defect prevention *a priori*) measures in future projects.

It has been recognized that better product models applicable early in the life cycle are needed. Examples include models of requirements (e.g., subjective measures to characterize the importance or complexity of requirements), designs (e.g., subjective measures to characterize the readability of designs and objective measures to characterize the complexity), and test plans (e.g., coverage and defect classes). The problem with product measures applicable before code has been developed is that the objects to be measured are not described in any measurable form. They are either not described at all (e.g., very often design decisions contributing to design complexity are not documented Rombach, 1984, 1990) or described informally (e.g., requirements are described in English prose, not even enabling the identification of individual requirements to judge their importance and complexity).

More recently, the importance of measurable process models has been recognized (Basili and co-workers, 1992; McGarry, 1985). Most attempts of measuring processes so far have been limited to high level life cycle models or implicit models of design, testing, reading, and maintenance processes. The problem with the measurement of implicit processes has been described (Rombach and Ulery, 1989). The process maturity work of Humphrey (1989) and others at the SEI has triggered an increased awareness of process, within both industry and academia. As a result, companies have started to document their processes explicitly (yet informally). In academia, a number of research groups have started to develop process representation formalisms (Kellner and Rombach, 1990; Osterweil, 1987; Rombach, 1991).

Example Product Models and Measures

The are a variety of software products, requirements documents, specification documents, design documents, source code, object code, test plans, user documentation, etc. For each such product, there are a variety of models that characterize some aspect of the product. These models offer visibility and insights into the product from a variety of perspectives. Product models can be broken down into several categories. Two major categories are static and dynamic models. Static models are based on the static properties or structure of the product, whereas dynamic models are based on the execution behavior of the product. Static models include size and structural complexity. Dynamic models include reliability, performance, and test coverage. Each of the models provides an insight into some property of the product.

Product models and measures can be used to evaluate the process and product; to estimate the cost and quality of the product; and as a feedback mechanism during software development and evolution, to monitor of the stability and quality of the product. They provide a quantitative view of the software development process and product, can be used to refine and engineer techniques and tools, can help with technology transfer, and can be used to provide quality assurance probes into the product to provide visibility to anyone who needs it. In their most refined form, they can be used in contracts for award, acceptance, and budget incentives. In the next sections, models and measures for size, control and data structure complexity, reliability, and test coverage will be discussed.

Size Models. Although size is the most common measure, it is the most difficult to articulate. That is because it depends on so many different factors, e.g., the notation, the formatting style of the notation, what aspects of the product are considered part of its size (e.g., are comments extraneous information or essential parts of the size of a product), etc. For example, if one is interested in the size of a requirements document, a great deal depends on whether the document is written in English or some formal notation and how requirements can be distinguished and enumerated. If the requirements are written in English and it is impossible to distinguish or enumerate independent requirements, then how does one model size? It is possible to count the pages of the document, which has been done, but now one must consider type font, spacing, etc.

The problems associated with models of size fall into two categories. The first is what is counted, ie, what models for size are used. These problems are partly addressed by the GQM in that the appropriate models to choose depend on the goals. Deciding what is to be counted, e.g., enumerated requirements and pages fall into this category. The second problem is associated with whether or not there are reasonable measures that can be associated with the model. Being able to distinguish requirements, ie, having a mechanism for enumerating requirements falls into this category. This problem depends on the ability to formalize the notation of the product in such a way that the measure makes sense.

Because size measures depend on the product and its model, some examples of size measures and the products they can be used on follow.

S1. Number of pages of the document.
S2. Number of requirements.
S3. Number of functions in the specification.
S4. Number of subsystems.
S5. Number of modules.
S6. Number of function points.
S7. Number of procedures, functions or packages.
S8. Number of lines of code.
S9. Number of tokens.

None of these measures is good or bad in its own right. It depends on how they are used, whether they satisfy the concept to be analyzed. The number of pages of documentation can be associated with any document. It is a very gross measure and highly dependent on the notation and page format. However, if one is interested in comparing products of similar notations and format is serves as a simple gross measure of size.

The number of requirements is an excellent measure of any document representing the function of the system, e.g., the requirements document, the specification, the design, and code. It provides a measure of the functionality that the system represents and can be used to compare documents of similar size functionality. For example two systems that have the same number of requirements (and are of a similar nature) yet have dramatically different fault rates or lines of code associated with them provide some insight into the development process used to produce the code. The problems associated with this measure are the ability to distinguish requirements, account for their interdependence, and choose the level at which to count.

For example, in a compiler, enumerating requirements as the number of declarations and statements that need to be translated is one mechanism for enumerating requirements. However, the definition of the statements and their interdependence in terms of the effect on the runtime environment clearly changes the concept of size if one is comparing the size of compilers for two languages that have very different runtime environments. Clearly a great deal depends on how the size measure is used. If it is used to show that languages with a similar number of requirements have very different designs and implementations based on the sizes of the implementation, then the measure of number of statements is a reasonable measure of the number of requirements. If it is used to justify that the compilers should both be the same size, then the measurer is being misused.

The number of functions in the specification is similar to the number of requirements, except it views functionality from the point of view of the designer rather than that of the user or customer. The number of subsystems is primarily a measure of the design, as is the number of, modules. However, both these definitions require careful definitions of the terms. As is seen in the literature, *module* can be used to mean everything from a Fortran subroutine to an information hiding subsystem.

The number of function points is a derived measure associated with the amount of data being processed. The number of procedures, functions or packages is a macromeasure of code size or a micro measure of design. It can be used for estimation of effort, because it is possible to categorize these units by application or type and associate a data base of effort at a detailed level of specificity.

The number of lines of code is a measure of code. There are a variety of measures that can be associated with the generic concept of lines of code. One can count total lines including comments, total lines excluding comments, executable statements, etc. All of these are useful measures, depending on why one is interested in counting lines of code. For example, most cost models total lines of source code, because there is an effort associated with producing everything, including comments. On the other hand, if one is interested in approximating functionality at the code level, the executable statements is probably a better measure. Once again, it depends on why one is measuring size.

The number of tokens is a micromeasure of the number of units of information. For example, if one is counting the number of tokens in a program, one might count the number of operators and operands in the source code. If one is counting the number of tokens in a requirements document written in English, one might count the number of nouns and verbs. In the next section, a theory of program measurement based on token count as suggested by Halstead (called software science) is discussed.

Software Science Measures. An approach to size measurement is to use the number of tokens as the basic syntactic and semantic units of size. Such an approach was developed by Halstead (1977) in an attempt to study the measurable properties of algorithms. In doing this he distinguished between the concept of an operator and an operand. The theory was developed for programs written in some programming language. Thus an operand was defined as a variable and an operator was defined as an arithmetic symbol, command name, (e.g., while and if), or any other form of function or procedure name. The basic measures are defined as follows.

n_1 =Number of unique or distinct operators in an implementation.

n_2 =Number of unique or distinct operands in an implementation.

N_1 =Total usage of all operators.

N_2 =Total usage of all operands.

$f_1 j$ =Number of occurrences of the jth most frequent operator, $j=1, 2, \ldots, n1$

$f2, j$ =Number of occurrences of the jth most frequent operand, $j=1, 2, \ldots, n2$

The vocabulary n of the algorithm or program is defined as

$$n=n1+n2$$

The implementation length, a measure of the size of a program, is

$$N = N_1 + N_2$$

and

$$N_1=\sum_{j=1}^{n_1} f_1 j$$

$$N_2=\sum_{j=1}^{n_2} f_2 j$$

$$N=\sum_{i=1}^{2}\sum_{j=1}^{n_i} f_2 j$$

Example: Euclid's Algorithm

```
IF (A=0)
LAST:                BEGIN GCD :=B;
   RETURN END;
IF (B=0)
   BEGIN GCD :=;
   RETURN END;
HERE:              G :=A/B;
   R :=A - B * G;
   IF (R=0)
      GO TO LAST;
```

```
A :=B;
B :=R;
GO TO HERE
```

Operator Parameters: Greatest Common Divisor Algorithm

Operator	j	f_{1j}
;	1	9
:=	2	6
( ) or BEGIN ... END	3	5
IF	4	3
=	5	3
/	6	1
-	7	1
	8	1
GO TO HERE	9	1
GO TO LAST	10	1
	n_1=10	N_1=31

Operand Parameters: Greatest Common Divisor Algorithm

Operand	j	f_{2j}
B	1	6
A	2	5
O	3	3
R	4	3
G	5	2
GCD	6	2
	n_2=6	N_2=21

There are a variety of measures derived from the basic measures $n1$, $n2$, $N1$, $N2$. Implementation length N is approximated as a function of the unique operators and operands based on a formula from information theory. This function, called program length is

$$N\,\hat{}\; = n_1 * \log_2 (n_1) + n_2 * \log_2 (n_2)$$

The program length represents the number of bits necessary to represent all tokens that exist in the program at least once. It can be considered as the number of bits necessary to represent the symbol table of the program.

Another measure, program volume, represents the size of an implementation,

$$V=N * \log_2 (n)$$

It can be thought of as the number of bits necessary to represent the entire program in its minimal form, ie, independent of token name length.

The meaured potential volume,

$$V^*=(2+n_2^* \log_2(2+n_2^*)$$

is an attempt to approximate the minimal number of tokens necessary to specify the algorithm. It represents the minimal number of input and output parameters and operators needed for a specification. It is based on the program volume equation with N defined as 2 operands, the function name and the bracketing operator ( ) for the parameters, plus $n2^*$, representing the number of operands in the program occurring only once each.

Using V and V^*, one can represent the measure program level, which is the level of an implementation,

$$L=V^*/V$$

The larger the program volume, the lower level the implementation. Thus an algorithm written in a low level language, requiring more volume, ie, repetition of the operands and a large number of operators, will have a lower level implementation than an algorithm implemented in a language that requires a smaller number of operators and smaller number of repetitions of operands.

It is possible to approximate L with L^, by the formula $(2 * n_2/ (n_1 * N_2)$ and define a meaure for program difficulty of implementing an algorithm in a particular languages as $D=1/L$. Programming effort E can then be defined as the volume times the difficulty. $E=V \times D=V/L=V_2/V^*$ It has been suggested that E might be thought of as representing the effort required to comprehend an implementation rather than to produce it. One can think of E as a measure of program clarity.

Cyclomatic Complexity. Based on the concept that the number of test cases is based on the number of linearly independent circuits in a program, McCabe suggested that the cyclomatic number might be a good approximation of the complexity of a program from the point of view of test effort. The cyclomatic number $v(G)$ of a graph G with n vertices and e edges, and p connected components is

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

In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits. Thus if a program is viewed as a graph that represents its control structure, the cyclomatic complexity of a program is equal to the cyclomatic number of the graph representing its control flow.

Cyclomatic complexity has several properties:

1. $v(G) >= 1$.
2. $v(G)$=maximum number of linearly independent paths in G; it is the size of a basis set of the set of paths through the program.
3. Inserting or deleting a functional statement to G does not affect $v(G)$.
4. G has only one path, if and only if $v(G)=1$.
5. Inserting a new edge in G increases $v(G)$ by 1.
6. $v(G)$ depends only on the decision structure of G.

It can be shown that the cyclomatic complexity of a program equals the number of predicate nodes plus 1. This provides an easy mechanism for calculating the cyclomatic complexity of a program:

$$v(G)=\text{number of decisions}+1$$

The concept of cyclomatic complexity is tied to testability, is intuitively satisfying in that it describes the complexity of a program in terms of the cost of testing the number of independent circuits in a program, and is easy to compute.

Segment–Global Usage Pairs. In an attempt to evaluate the usage of global data for a program, Basili and Turner (1975) used a measure, to study the quality of the use of globals from the point of view that if a variable is declared global, it should be used by most of the segments to which it is available.

A segment-global usage pair (p,r) is an instance of a global variable r being used by a segment p, i.e., r is either modified or set by p. Each usage pair represents a unique Ruse connection between a global and a segment.

Let AUP, represent the count of the actual usage pairs, i.e., r is actually used by p. Let PUP represent the count of potential usage pairs, i.e., given the program's globals and their scopes, the scope of r contains p so that p could potentially modify r. This represents the situation in which every segment uses every global. Then the relative percentage usage pair (RUP) is

$$\mathrm{RUP}=\mathrm{AUP}/\mathrm{PUP}$$

which represents a way of normalizing the usage pairs relative to the program structure. The RUP measure is an empirical estimate of the likelihood that an arbitrary segment uses an arbitrary global.

The measure RUP lies between 0 and 1. If RUP is small it means that even though a large number of variables were defined as globally available, only a few of them were really used by the segments to which they were available. This implies there may be something wrong with the hierarchical structure of the data in that there appears to be less cause for making the variables global. If RUP is close to 1, it means that those variables that were defined as global were really used by those segments to which they were made available. Thus the hierarchical structure of the data is truly representative of their use. This measure depends on the programming language and what kind of hierarchical data structuring it allows.

Data Bindings Measure. Myers (1978) has argued that modules should be formed in a manner that reduces coupling and increases strength. In an effort to measure the data coupling between modules, Basili and Hutchens (1983) used the concept of data bindings to capture the coupling relationship between program segments.

A segment–global–segment data binding (p,r,q) is an occurrence of the following:

1. Segment p modifies global variable r.
2. *Variable* r is accessed by segment q.
3. p and q are different.

The existence of a data binding (p,r,q) implies q depends on the performance of p because of r. Binding (p,q,r) is not the same as binding (q,r,p). Binding (p,r,q) represents a unique communication path between p and q. The total number of data bindings represent the degree of a certain kind of connectivity, i.e., between segment pairs via globals, within a complete program.

Letting ADB (actual data bindings) represent the absolute number of true data bindings in the program, i.e., the true connectivity, and PDB (possible data bindings) represent the absolute number of potential data bindings given the program's global variables and their declared scope, they defined RDB (relative percentage of data bindings) as RDB=ADB/PDB, the normalized number of data bindings within the program.

One problem with the data bindings measure is that is does not take into account indirect bindings, through parameters, and the calling program. Thus implementations of an abstract data type may show no explicit bindings, even though the program elements are tightly bound. An example of using data binding metrics for the purpose of characterizing the reuse potential of Ada components is given in Basili and co-workers, 1988.

Span. In looking for a measure of the complexity of understanding a program, Elshoff defined the concept of data span as a basis for the amount of knowledge (number of variables) one needs to be aware of at any point in a program (Elshoff, 1977). Span is the number of statements between two textual references to the same identifier. Thus in the program segment

```
X := Y ; Z :=Y ; X :=Y T
```

Span (X)=count of number of statements between the first and last statements (assuming no intervening references to X). Y has two spans. For n appearances of an identifier in the source text, $n-1$ spans are measured. All appearances are counted except those in declare statements. If the span of a variable is greater than 100 statements, it implies that one item of information must be remembered for 100 statements until it is used again.

Using span, complexity can be defined as either the number of spans at any point (using either the maximum, average, or median of that number as the actual value) or the number of statements a variable must be remembered, on the average (e.g., average span over all variables). One potential variation of this measure is to perform a live–dead variable analysis and then define the complexity of a program to be proportional to the number of variables alive at any statement.

One way to scale this measure up for a complexity measure for a module M might be by using the following formula:

$$C(M)=\sum_{i=1}^{\textit{number of statements}} \times \; (n_i * s(n_i))/(\textit{number of statements})$$

where ni=the number of spans of size $S(n_i)$. In a study performed by Elshoff, variable span has been shown to be a reasonable measure of complexity. For commercial PL/1 programs, he showed that a programmer must remember approximately 16 items of information when reading a program. This argues for the difficulty of program understanding.

Example Process and Project Models and Measures

There are a variety of reasons for modeling resource. We may wish to do an initial prediction of resources, i.e., based upon a set of factors that can be estimated about a project, we can try to predict total effort, cost, staffing, computer use, etc.

We can develop descriptive models of the development pattern, that is show how and when resources get allocated. This provides insights into what is going on, shows how different parameters change the resource allocation pattern, helps us evaluate the effects of various techniques and methods so we can better engineer software, and provides baselines from which to plan future developments.

We can use resource modeling to help predict the resources to be used in the next phase based upon the current phase. This is a more detailed application of the initial prediction activity mentioned above. That is, given where we are in the project, the model should tell us what should happen next. If it does not, why not; is it a sign of trouble, etc.?

Lastly we can use the model for validation of our understanding of the environment, i.e., does the model explain our behavior and environment, do the factors (parameters) agree with our environmental factors, and are they calibrated correctly?

A good model explains our behavior and the development environment. Its parameters are calculable from known or easy to estimate data, e.g., maximum staffing, time to delivery, complexity of the software, number of lines of code, number of modules, number of I/O formats, type of software, amount of old/new software (design, code, specification). The parameters describe and can be calibrated for our environment, redundancy checks and risk analysis factors are available and when the model does not work, we can gain insight into why and what is different about the environment than was expected or how to improve the model or better calibrate it to the environment.

There are a variety of resource models that provide descriptions of such resources as effort, staffing, cost, computer use, and calendar time. These models can be categorized with respect to type of formula used for total effort. Models can be distinguished as single variable vs. multivariable, empirically based on theoretically based. Some describe the dynamic allocation of resources while others give a single overall figure. Some are defined at the macro level, based on high level parameters while others are defined at the micro level.

Early on Barry Boehm collected data to provide some insights into where the effort resources were being spent. He surveyed several projects and collected data within his own organization, TRW. Based upon his data, the 40-20-40 rule-of-thumb came into practice, ie 40% of the resources should be spent on analysis and design, 20% on code, and 40% on checkout and test. However, this is a misinterpretation of the data. The data represents phase data data, rather than actual resource expenditures, ie, a better interpretation of the data might be that 40% into the resources the design completion milestone should be reached, and 60% into the resources, the code completion data should be reached. This provides data for management as to when various milestones are appropriate.

Table 1 provides similar data from IBM and the SEL at NASA Goddard Space Flight Center as a basis for comparison. In this data, another category, other, is present. Other represents resource effort that is not associated with a particular phase of the life cycle, e.g., training, meetings, etc.

Examining the first three columns, it should be noticed that the data from the three environments are different, ie the design effort varies from 40% to 35% to 20%, code effort varies from 20% to 30% to 45%, and test effort varies from 40% to 25% to 28%. Only two organizations report effort in the other category. There are several explanations for this. First, each of the organizations may allocate different activities to the phases, e.g., in the SEL, the life cycle starts with functional specification analysis and design, which are what analysis and design mean. Second, each organization may have a different point at which they define their milestones, e.g., the SEL defines the analysis and design phase to end at PDR (preliminary design review). Clearly, the interpretation of the data is highly dependent upon the environmental characteristics in which it was collected. That makes it very difficult to compare across environments.

It should also be noted that there are two sets of data for the SEL environment. That is because two types of data are collected, phase date data and activity. The phase date data represents the total amount of effort expended each week by each staff member within the dates estabished for the phase. The activity data is the total amount of effort expended on each activity by each staff member each week. These figures are not the same because staff member perform activities in phases other than the named phase they are in. For example, coding may begin on some components before the design end milestone occurs. Also, design may continue to occur after the design end milestone because the design may not be complete or some redesign may be necessary. Activity patterns do not necessarily follow date patterns. Both sets of data are important. The activity data provides insight into the actual effort expended in each activity. The phase data provides insight into how management should allocate milestones.

The other category in the SEL activity column is 27%. This says that project staff spend 27% of their time performing activities chargeable to the project but not directly

Table 1. Where Does the Software Effort Go?

	Analysis and Design	Coding and Auditing	Checkout and Test
Sage	39%	14%	47%
NTDS	30	20	50
Gemini	36	17	47
Saturn V	32	24	44
S/360	33	17	50
TRW Survey	46	20	34

associated with a project activity, ie, travel, education, meetings, etc. Actually the figure for the SEL is estimated as low compared to what that figure might be in other organizations. It is worth noting that if one estimates solely on the basis of activities, the estimation would be 27% low in the SEL (Table 2).

Once an effort estimate is made, the next question is how to assign people to the project so that the deadlines for the various development activities will be met. Each of the methods discussed so far uses an empirical approach to identify the activities that are parts of the development process of a typical project within their environment. Using effort data from past projects, the percentage of effort expended on each activity is estimated is determined. These percentages serve as a baseline and are intuitively adjusted to meet the expected demands of a new project. For example, total cost may be allocated into five major subareas: analysis cost, design cost, coding cost, testing cost, and documentation cost. Each of these subareas is subdivided again, depending upon the activities in the subareas. In this way, each activity can be staffed according to its individual budget. Allocation of time is determined by history and good management intuition.

An alternative approach is to justify resource expenditures based upon an underlying theory of how people solve problems. We will refer to these types of approaches as theoretical dynamic resource models. The original model of this type is due to Larry Putnam.

The model is based upon a hardware development model (due to Peter Norden) which noted that there are regular patterns of staff buildup and phase-out independent of the type of work being done. It is related to the way people solve problems. Each activity could be plotted as a curve which grows and then shrinks with regard to staff effort across time. Norden isolated several activities associated with hardware development: planning and specification, design, prototyping, and release. Similar curves were derived by Putnam for software cycles: planning, design and implementation, testing and validation, extension, modification and maintenance.

The basic theory behind this model is based upon the ideas that software development is a problem-solving effort, design decision making is the exhaustion process, and activities partition problem space into subspaces corresponding to the various stages (cycles) in the life cycle. Assumptions about the problem subset are that the number of problems to be solved is finite, the problem-solving effort makes an impact on and defines an environment for the unsolved problem set, a decision removes one unsolved problem from the set (the model assumes events are random and independent) and the number of people is proportional to the number of problems "ripe" for solution.

Based upon these assumptions, Putnam derives an effort curve, the integral form of the life cycle equation

$$y=K^*(1-e^{-at})^2$$

where

y=the cumulative manpower used through time t

K=the total manpower required by the cycle stated in quantities related to the time period used as a base, e.g., staff-months/month

a=A parameter determined by the time period in which y reaches its maximum value (shape parameter)

t=time in equal units counted from the start of the cycle

The life cycle equation (derivative form) is

$$y^1=2\,K\,a\,t\,e^{-at^2}$$

where

y=the manpower required in time period t stated in quantities to the time period used as a base

K=total manpower required by the cycle stated in the same units as y

The curve represents the staffing buildup (a Rayleigh curve). Putnam argued that the sum of the individual cycle curves results in a pure Rayleigh shape because software development is implemented as a functionally homogeneous effort (single purpose).

The shape parameter a depends upon the point in time at which y reaches its maximum, i.e.,

$$a=1/2t_d^2$$

where t_d is the time to reach peak effort. Putnam empirically showed that t_d corresponds closely to the design time (time to reach initial operational capability). Substituting for a we can rewrite the life cycle equation as

$$y^1=\frac{K}{t_d^2} * te^{-t2/2td2}$$

Based upon his analysis, Putnam concludes that large software development projects follow a life cycle pattern describable by the Rayleigh (manpower) equation:

$$y^1=2\,K\,a\,t\,e{-at^2}$$

Software systems have 3 fundamental parameters: the life cycle effort (K), development time (t_d), and difficulty ($D=K/t^2$). Productivity is related to the difficulty and state of technology constant C_k. Management cannot arbitrarily increase productivity. Management cannot reduce development time without increasing difficulty. The tradeoff law shows the cost of trading time for people.

Table 2. **Comparison of Resource Effort Data**

	TRW	IBM	SEL Phase	SEL Activity
Design	40	35	20	21
Code	20	30	45	28
Checkout/Test	40	25	28	23
Other		10	5	27

RESOURCE PLANNING

We have discussed only a sampling of the models that exists in the literature. There are a variety of other models, but most of them are of a similar to those discussed above.

In order to estimate resource, the models should be an aid to software development management and engineering. Common sense should never be abandoned.

None of these models are not accurate enough to be taken as the sole source. One approach to doing resource estimation might be to apply more than one model and examine the range of predictions offered, compare the results. If they agree, one can be more secure about the estimate.

If they do not agree, examine why not, ask what model assumptions were not satisfied and what makes this project different. Make sure you are comfortable with the explanation of the difference.

What is needed is a system that combines global and local of models, allowing for different inputs at different times, and providing an integration mechanism, that allows us to better understand the parameters of both sets of models. This way we can learn from our application of the system so that we can improve our estimates over time. That is, if we apply both global and local models when possible, and based upon our analysis of the both models, modify our understanding of their parameters and evolve them with respect to the choice of baseline equation and cost drivers to provide better accuracy for future projects, we can do a better job of cost estimation and comparison with other environments in the future.

In order to provide a basis for resource planning and allocation and take advantage of prior history, Jeffrey and Basili prosed a resource consumption model that characterized types of resources and their use.

Resource are consumed during the software process. Software process characteristics are superordinate to the resources consumed on a project. A process characterization includes such characteristics as project type, organizational development conventions, project manager preferences, target computer system, development computer system, project schedules or milestones, and project deliverables.

They classify a variety of different types of resources: hardware, software, human, calendar time, support, e.g., supplies, materials, communications, facility costs, etc. They categorize the viewing of resources by three different dimensions: incurrence, availability, and use descriptors. Incurrence is subdivided into estimated and actual. Availability is subdivided into desirable (resources of value), accessible (resources able to be used), and utilized (resources used). Desirable resources are those considered ideal for any project, i.e., unconstrained by availability, implying the ideal hardware, software and calendar time, ideal people characteristics. Accessible resources are theoretically those available to the project available within the organization, chosen from data base of corporate resources. Utilized resources are those available and anticipated to be used or actually used by the project, i.e., those driven by project constraints (e.g., cost) and other corporate needs. Use Descriptors are subdivided into the nature of the work or activity (e.g., testing, design), the point in time (e.g., calendar dates needed), and the resources utilized (e.g., hours, dollars, units).

Information about resources can be obtained from individual and group knowledge, a knowledge base, a data base of prior projects, or algorithmic models. Inputs to the model can occur at project milestones, by manager initiated points in time based upon divergence between estimated and actual, or at system initiated points in time based upon divergences recognized by the measurement system.

If we combine incurrence and availability dimensions, we get the following categories: Estimated Desirable: those resources considered ideal for the project at planning time, Estimated Accessible: those resources available with the organization that can used by the project, Estimated Utilized: those resources anticipated to be used by the project, Actual Utilized: those resources actually used by the project, Actual Accessible: with hindsight the resources which were available and should have been utilized, Actual Desirable: with hindsight the resources which should have been made available and used.

The differences between each of the different resource models provides a unique form of input to the organization. The difference between estimated desirable and estimated accessible provides input to the risk management plan. The difference between estimated accessible and estimated utilized provides input to the contingency plan. The difference between the estimated and actual utilized provides input to the manager for real time adjustments to resource allocation and can be used to provide measures of progress and problems. The differences between actual utilized and accessible is feedback needed for future project planning, ie, what should we really have used to make this project work. It represents an analysis and smoothing of the actual utilized resources. The difference between actual accessible and actual desirable is the feedback to corporate planning, ie, what resources does the corporation have to acquire in order to complete projects of the type developed successfully.

When viewed from the point of view of the Improvement Paradigm, resource planning involves the following activities: During planning, decisions are made about such things as obtaining a further resource (updating the corporate resource data base), committing to development without the expert, negotiating for full or partial decommitment of the expert. During execution, review and re-estimation are done to modify the plan by allocating contingency resouces, revise estimates of accessible resources, and revise desirables. During post mortem analysis and the packaging of experience, revisions are made to various experience base models, including project and environment models, and the desirable and accessible resources models. Lessons learned are developed.

Validation of Measures

There have been a large number of studies of these measures, most of which show the correlations among lines of code, $v(G)$ and E. Some of these studies show limited relationships between the measures and effort and faults.

Typical of the results is an early study on evaluating and comparing several of the software measures in the literature in which Basili, Selby and Phillips (1983) posed the following questions:

- Do measures like cyclomatic complexity and the software science measures relate to effort and quality?
- Does the correspondence increase with greater accuracy of data reporting?
- How do these measures compare with traditional size measures such as number of source lines or executable statements?
- How do these measures relate to one another?

They defined effort and quality based upon available data in the SEL. Effort was defined as the number of manhours programmers and managers spent from the beginning of functional design to the end of acceptance testing. Quality was defined as the number of program faults reported during the development of the product.

The data used in the analysis was commercial software. The application domain was satellite ground support systems consisting of 51,000 to 112,000 lines of FORTRAN source code. Ten to 61% of source code was modified from previous projects. The development effort ranged from 6,900 to 22,300 man hours. The analysis focused on data from 7 projects, only newly developed modules, i.e., subroutines, functions, main procedures and block data's.

Measures studied included:

Source lines of code

Source lines of code excluding comments.

Executable statements.

Software science measures.

N : Length in operators and operands.

V : Volume.

*V** : Potential volume.

L : Program level.

E : Effort.

B : Bugs.

Cyclomatic complexity.

Cyclomatic complexity excluding compound decisions (referred to as cyclo _cmplx _2).

Number of procedure and function calls.

Calls plus jumps.

Revisions (versions) of the source in the program library.

Number of changes to the source code.

Figure 1 show the relationship of the measures to actual effort. It should be noted that because the data come from real projects, there is the potential for error in the reported effort data. However because project data are reported in two forms, a reliability check was performed and the data are rated by ensurance of their accuracy, given as a percent.

The data show that there is some relation between the measures and the effort data across all projects. However the relation improves with individual project data, validated data, and individual programmers data. Note that the Spearman correlations (*R* values) are all significant at $p=0.001$ level. Figure 2 shows the relationship between the measures and the number of faults found in the modules. The number of program faults for a given module is the number of system changes that listed the module as affected by an error correction. To differentiate among the various faults, faults were weighted by the effort expended. Weighted faults (*W_flts*) is a measure of the amount of effort spent isolating and fixing faults in module. Note that the Spearman correlations are all significant at $p=0.001$, except for those marked with an *, which were significant at $p=0.05$. Note that when compared with faults, the relations are low overall with the number of revisions showing the strongest relationship. However, as before, the relations improve with individual projects or programmers.

Figure 3 shows the relationships among the various complexity measures. There were 1794 modules considered in this study and the Spearman correlations are all significant at $p=0.001$. Here one may notice that many of the size and complexity measures correlate quite closely with one another. This is especially true for the various lines of code measures, cyclomatic complexity, and the software science measures. Thus there is an indication that they may be measuring the same thing. It should be noted, however, that revisions and calls do not correlate as well with the other measures or with each other.

	Dimension				
	All Projects	Single Project			Single Programmer
Validity Ratio	**all**	**all**	**80%**	**90%**	**92.5%**
#Modules	**731**	**79**	**29**	**20**	**31**
E^	.49	.70	.75	.80	.79
Cyclo_complx_2	.47	.76	.79	.79	.68
Calls/jumps	.49	.78	.81	.82	.70
Source lines	.52	.69	.67	.73	.86
Execut. stmts	.46	.69	.71	.78	.75
V	.45	.66	.69	.77	.72
Revisions	.53	.68	.72	.80	.68

Figure 1. Measures' relationships to actual effort.

	Dimension					
	All Projects		Single Project		Single Programmer	
#Modules	**652**		**132**		**21**	
	faults	**w_flts**	**faults**	**w_flts**	**faults**	**w_flts**
E^	.16	.19	.58	.52	.67	.65
Cyclo_cmplx_2	.19	.20	.55	.49	.48*	.45*
Calls & jumps	.24	.25	.57	.52	.60*	.56*
Source lines	.26	.27	.65	.62	.66	.65
Execut. stmts	.18	.2	.54	.51	.58*	.53*
B	.17	.19	.54	.50	.68	.66
Revisions	.38	.38	.78	.69	.83	.81
Effort	.32	.33	.64	.62	.67	.62

Figure 2. Measures' relationships to program faults.

Based on performing this study in a commercial environment, the authors concluded that one could use commercially obtained data to validate software measures but that validity checks and accuracy ratings are useful. The strongest effort correlations are derived with modules from individual programmers or certain validated projects and the majority of effort correlations increase with the more reliable data.

In regard to the ability of the measures to predict effort or quality, the authors concluded that none of the measures seems to explain satisfactorily effort or development faults and that neither software science's *E* measure, cyclomatic complexity, nor source lines relates convincingly better with effort than the others. The number of revisions correlates with development faults better than either software science's *B* measure, *E* measure, cyclomatic complexity or source lines of code. It was also concluded that many of the size and complexity measures relate well with each other, which might mean they are measuring the same thing.

Automatable Measures.

A variety of measures that can be taken on the product have been examined. Most of these can be automated. A measure can be considered automatable if there is no interference to the developer, it is computed algorithmically from quantifiable sources, and it is reproducible on other projects with the same algorithms. There are a variety of measures, besides those measures discussed so far, that are automatable.

The trouble with many automated measures is that they are indirect measures, i.e., they measure some aspect of the process or product that may not be directly measured. The hope is that they represent or correlate with something of deeper interest. For example, in the earlier study, there was some relationship between effort and size, i.e., a measure of size provided some indication of the effort and as size grew, so did effort. Furthermore, the version number gave some indication of the number of defects in the same way. Although version number itself can be easily calculated, it is not a direct measure of defects, in fact, in some environments, it may not even correlate with defects. For example, if in a particular environment, new versions are created only after a series of changes have been made, then one might not expect to see a relationship between defects and version numbers.

An automatable measure is considered useful, if it is sensitive to externally observable differences in the development environment and the relative values correspond to some intuitive notion about characteristic differences in the environment. Thus version number, in the earlier study might be considered useful in that is it sensitive to observable differences, e.g., the number of defects found in the product, and it corresponds to the fact that as the

	Source Lines (SLOC)	Revisions	Calls & Jump	Calls	Cyclo-Cmplx_2	Cyclo-Cmplx	Exec.Stmts	SLOC Commts
E	.83	.37	.89	.62	.89	.88	.95	.86
V	.82	.35	.87	.57	.87	.87	.96	.86
SLOC - Cmmts	.93	.49	.88	.68	.86	.85	.91	
Execut Stmts	.85	.38	.91	.61	.92	.91		
Cyclo-Cmplx	.81	.39	.95	.55	.99			
Cyclo-Cmplx_2	.82	.38	.94	.56				
Calls	.66	.41	.75					
Calls & Jumps	.85	.44						
Revisions	.50							

Figure 3. Relationships among the measures.

number of defects increases, so does the version number. However, this usefulness of an automated measure must be checked for the environment in which it is being used.

Examples of automatable measures include program changes (Dunsmore and Gannnon, 1980), textual revision in the source code representing one conceptual change, and job steps (Basili and Reiter, 1979), the number of computer accesses during development or maintenance. More specifically, program changes are defined as textual revisions in the source code of a module during the development period, such that one program change should represent one conceptual change to the program. A program change is defined as one or more changes to a single statement, one or more statements inserted between existing statements, or a change to a single statement followed by the insertion of new statements. The following are not counted as program changes: the deletion of one or more existing statements, the insertion of standard output statements or special compiler-provided debugging directives, or the insertion of blank lines or comments, the revision of comments and reformatting without alteration of existing statements. Program changes have been shown to correlate well with defects in a particular environment (Dunsmore and Gannon, 1980).

Job steps are defined as the number of computer accesses representing a single programmer-oriented activity performed on the computer at the operating system command level. Examples include text editing, module compilation, program compilation, link editing, and program execution. Job steps are basic to the development effort and involve nontrivial expenditures of computer or human resources and were shown to correlate with human effort in a particular environment.

MEASUREMENT METHODOLOGY

A comprehensive Software Measurement Methodology needs to be embedded into an improvement-oriented software engineering paradigm aimed at achieving higher levels of software quality in a controlled way. In such context, measurement can be used in three stages:

- At the beginning of an improvement program: software measurement provides a way to establish a baseline for current quality of products and processes, and to a plan for the tasks that enact the program.
- During the execution of a task within an improvement program: software measurement provides the necessary control over the execution of the task and possible corrective actions.
- After the execution of task within an improvement program: software measurement provides a way to evaluate whether the assigned goals have been achieved or not.

Once again we see the necessity of associating measures and goals in order to take full advantage of the power of measurement. In the article on the Goal Question Metric Approach we present a way to develop such association. The overall methodological framework, however, is presented in the article on the experience factory, which also discusses the organizational issues. In synthesis, the methodology we propose is based on the Quality Improvement Paradigm, which consists of six fundamental steps:

1. Characterize the current project (or project segment) and its environment using metrics and models.
2. Set the quantifiable goals for successful performance or improvement in the specific project (or project segment).
3. Choose the appropriate process model and supporting methods and tools.
4. Execute the processes, construct the products, collect and validate the measures, and analyze them in order to provide real-time feedback for corrective action.
5. Analyze the measurement data to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements.
6. Package the experience in the form of updated and refined models (or other forms of structured knowledge) and save it for future reuse.

Beside the Quality Improvement Paradigm there are a variety of methodological frameworks for using measurement in a quality improvement process (Basili, 1993). Plan/Do/Check/Act, the well-known Deming Cycle, is a quality improvement framework based upon a feedback cycle for measuring and optimizing a single process model. Total Quality Management (TQM) is a quality improvement framework based on measurement of customer satisfaction and enhanced communication within the organization. Lean Enterprise Management focuses on value added activities and measures the performance of the organization on those activities.

CONCLUSIONS

Lord Kelvin's statement that "one does not understand what one cannot measure" is at least as true for software engineering as it is for any other engineering discipline. Measurement has been recognized as an indispensible prerequisite to introducing engineering discipline to the development, maintenance and use of software products. The scope of measurement has matured from a set of measures to a set of techniques and methods aimed at supporting a large variety of software related goals via measurement. It has been demonstrated in local environments that sound measurement programs can lead to significant degrees of engineering control and can be the basis for controlled technology improvements. As an example, we refer the reader to publications from the NASA/SEL environment (Basili and co-workers, 1992; McGarry, 1985), where impressive results regarding the low-variance prediction of productivity and quality characteristics as well as the introduction of formal reading techniques and Cleanroom development processes are reported.

Software and its development processes are complicated because of their changing characteristics from project to project as well as organization to organization. Software development is not "production". Therefore, software measurement is not simply a question of listing standard measures. Instead, measures have to be chosen, customized and used according to goals of interest. This article is based on such a nonproduction view of software development. We have tried to capture the historical development of the evolving discipline of software engineering measurement, characterize the current state of the art and practice, and point out future directions.

One of the major lessons learned from the history of software measurement is that the usefulness of software measures cannot be judged out of content. Only a goal of measurement determines the appropriateness of measures. As a result, academic and practical measurement activites have shifted towards measurement methodologies (see GOAL/QUESTION/METRIC PARADIGM), concerns of introducing measurement into real environments, and the feedback mechanisms (via explicit models) to enable learning within projects as well as across project boundaries (see EXPERIENCE FACTORY).

BIBLIOGRAPHY

V. R. Basili, "Data Collection, Validation, and Analysis," in *Tutorial on Models and Metrics for Software Management and Engineering,* IEEE Computer Society Press, 1980a, pp. 310–313.

V. R. Basili, "Resource Models," in *Tutorial on Models and Metrics for Software Management and Engineering,* IEEE Computer Society Press, 1980b, pp. 4–9.

V. R. Basili, *Software Development: A Paradigm for the Future,* paper presented at the Thirteenth Annual International Computer Software & Applications Conference, Orlando, Fla., Sept. 20–22, 1989.

V. R. Basili, "The Experience Factory and its Relationship- to Other Improvement Paradigms," *European Conference on Software Engineering 1993,* Keynote Address.

V. R. Basili and co-workers, *The Software Engineering Laboratory— An Operational Software Experience Factory,* Fourteenth International Conference on Software Engineering, Melbourne, Australia, May 1992.

V. R. Basili and D. H. Hutchens, "An Empirical Study of a Syntactic Complexity Family," *IEEE Transactions on Software Engineering* **9**(6), 664–672 (Nov. 1983).

V. R. Basili and R. W. Reiter, "An Investigation of Human Factors in Software Development," *IEEE Comput. Magazine,* 21–38 (Dec. 1979).

V. R. Basili and H. D. Rombach, *Tailoring the Software Process to Project Goals and Environments,* paper presented at the Ninth International Conference on Software Engineering, Monterey, Calif., Mar. 30–Apr. 2, 1987.

V. R. Basili and H. D. Rombach., "Implementing Quantitative SQA: A Practical Model," Guest Editor's Introduction, *IEEE Software* (Sept. 1987).

V. R. Basili and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.* **SE-14**(6), 758–773 (June 1988).

V. R. Basili and H. D. Rombach, "Support for Comprehensive Reuse," *Software Eng. J.,* 303–316 (Sept. 1991).

V. R. Basili, H. D. Rombach, J. Bailey, and A. Delis, *Ada Reusability Analysis and Measurement,* paper presented at the Sixth Symposium on Empirical Foundations of Information and Software Sciences, Atlanta, Ga., Oct. 19–21, 1988.

V. R. Basili and A.J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Transactions on Software Engineering* **SE-1**(4) (Dec. 1975).

V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Eng.* **SE-10**(6), 728–738 (Nov. 1984).

L. Belady and M. M. Lehman, "A Model of Large Program Development," *IBM Sys. J.* 3, 225–252 (1976).

B. W. Boehm, *Software Engineering Economics* Prentice-Hall, 1981.

B. W. Boehm, J. R. Brown, and M. Lipow, *Quantitative Evaluation of Software Quality,* paper presented at the Second International Conference on Software Engineering, 1976.

B. Curtis, H. Krasner, V. Shen, and N. Iscoe, *On Building Software Process Models under the Lamppost,* paper presented at the Ninth International Conference on Software Engineering, Monterey, Calif., Mar. 30–Apr. 2, 1987.

W. E. Deming, *Out of the Crisis,* MIT Center for Advanced Engineering Study, MIT Press, Cambridge, 1986.

H. E. Dunsmore and J. D. Gannon, "Analysis of the Effects of Programming Factors on Programming Effort," *J. Sys. Software,* 1, 141–153 (1980).

J. L. Elshoff, "The Influence of Structured Programming on PL/I Program Profiles," *IEEE Transactions on Software Engineering* **SE-3**(5) 364–368 (Sept. 1977).

M. Halstead, *Elements of Software Science,* Elsevier Science Publishing Co., Inc., New York, 1977.

S. Henry and D. Kafura, "Software Structure Metrics based on Information Flow," *IEEE Trans. Software Eng.* **SE-7**(5), 510–518 (Sept. 1981).

W. S. Humphrey, *Managing the Software Process,* Addison-Wesley Publishing Co., Inc., Reading, Mass., 1989.

IEEE, *IEEE Standard Glossary of Software Engineering Terms,* IEEE-Std-729-1983, IEEE, 1983.

M. I. Kellner and H. D. Rombach, *Comparisons of Software Process Descriptions,* paper presented at the Sixth International Software Process Workshop, Hakodate, Japan, Oct. 1990.

M. Kogure and Y. Akao, "Quality Function Deployment and CWQC in Japan," *Quality Prog.,* 25–29 (Oct. 1983).

T. J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.* **SE-2**(4) (1976).

J. A. McCall, P. K. Richards, and G. F. Walters. *Factors in Software Quality,* RADC, TR-77-369, Rome Air Development Center, Griffiss Air Force Base, N.Y., Nov. 1977.

F. E. McGarry, *Recent SEL Studies,* paper presented at the Tenth Annual SE Workshop, NASA Goddard Space Flight Center, Greenbelt, Md., Dec. 1985.

G. J. Myers, *Composite/Structured Design,* Van Nostrand Reinhold, New York, 1978.

J. D. Musa. "Software Reliability Measurement," *J. Sys. Software,* 1, 223–241 (1980).

L. Osterweil, *Software Processes are Software Too,* paper presented at the Ninth International Conference on Software Engineering, Monterey, Calif., Mar. 30–Apr. 2, 1987.

H. D. Rombach, *Software Design Metrics for Maintenance,* paper presented at the Ninth Annual SE Workshop, NASA Goddard Space Flight Center, Greenbelt, Md., Nov. 1984.

H. D. Rombach, "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Trans. Software Eng.* **SE-13**(3), 344–354 (Mar. 1987).

H. D. Rombach, "Software Design Measurement: Some Lessons Learned," *IEEE Software Mag.* **7**(2), 17–25 (Mar. 1990).

H. D. Rombach, *MVP-L: A Language for Process Modeling in-the-Large,* CS-TR-2709, University of Maryland, Department of Computer Science, June 1991.

H. D. Rombach, V. R. Basili, and R. Selby, eds., "Experimental Software Engineering Issues," *Proceedings of the International Workshop, Dagstuhl,* Sept. 1992, Springer-Verlag, Lecture Notes Series, 1993.

H. D. Rombach and L. Mark, *Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases,* paper presented at the Twenty-second Annual Hawaii International Conference on Systems Sciences II, Kona, Hawaii, Jan. 1989.

H. D. Rombach and B. T. Ulery, "Improving Software Maintenance through Measurement," *IEEE Proc., 581–595 (Apr. 1989).*

H. D. Rombach and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned from the SEL,* paper presented at the IEEE Conference on Software Maintenance, Miami Beach, Fla., Oct. 16–19, 1989.

R. W. Selby, V. R. Basili, and T. Baker, "CLEANROOM Software Development: An Empirical Evaluation," *IEEE Trans. Software Eng.* **SE-13**(9), 1027–1037 (Sept. 1987).

R. W. Selby, A. A. Porter, D. C. Schmidt, and J. Berney, *Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development,* paper presented at ICSE 91, Austin, Tex., May 1991.

T. Sunazuka and V. R. Basili, *Integrating Automated Support for a Software management Cycle into the TAME System,* CS-TR-2289, University of Maryland, Department of Computer Science, College Park, Md., July 1989.

H. Zuse, *Software Complexity,* Walter de Gruyter, Berlin, 1991.

VICTOR R. BASILI
GIANLUIGI CALDIERA
University of Maryland
H. DIETER ROMBACH
Universität Kaiserslautern