SOFTWARE METRICS

Software metrics as measures of the quality of software play an important role in the field of software engineering. Better understanding of the development process and the principles of sound software design and the ability to better estimate costs and effort of future projects are just several of the potential benefits of using metrics. In this article, some important features of software metrics are described, and we review the state of the art. Issues covered include the importance of validating a metric and performing supporting experiments, of developing and evaluating a quality model for software factors such as maintainability, of identifying the characteristics of software complexity, and the ever-increasing role of supportive data collection tools in the field of software metrics.

INTRODUCTION

According to Fenton (1), *measurement* can be defined as the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules. As such, measurement pervades everyday life, and continues to affect us in the way we live more and more. Indeed, the importance of measurement can be traced back through the centuries and the invention of the Kelvin scale of temperature measurement by Lord Kelvin, who said:

> When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind.

In the field of software engineering, software metrics help us to identify and understand various features of software products and processes. The application of measurement theory and practices to the software process provides a means of better understanding the world of software engineering. For example, attributes such as maintainability may be the focus of measurement investigations. This is particularly important in view of the software crisis, characterised by software projects which run over budget and over time. In this article, prominent areas of software metrics are examined, and the state of the art in terms of progress in these areas is described. A number of themes relevant to the area of software metrics are discussed, together with current progress being made in each. The second section of this article addresses the motivation for using software metrics. The role of measurement theory in establishing a theoretical foundation is discussed, together with the role quality models play in satisfying common goals of the software development process.

Establishing that any proposed metric is theoretically valid requires that it conforms to certain mathematical properties. In the third section 3, these properties are explained, and an example is given to show how, in practice, this validation works.

Theoretical validation of a metric must be supported by empirical evaluation of that metric, for example, determining which features of the proposed metric are most useful to the software engineer. In the next section, a description of the empirical software engineering field is given, identifying recent work in the area, and benefits obtainable from empirical analyses.

The difficulty of maintaining legacy software is a recognized problem in the software industry, and places a large burden on the software budget. The lack of a clear understanding of what constitutes quality software contributes to this phenomenon. Very little thought goes into understanding the constituents of software, the implications of ignoring features of software which have the most impact on the delivered system, and ways of improving software process management (2). The fifth section of this article addresses the problems of developing and evaluating rigorous quality models which address some of the problems mentioned. As the term suggests, a model should provide a means by which high-level external attributes can be decomposed into low-level internal attributes, which can be measured. A structure is thereby imposed throughout, ensuring a rigor in the analysis of quality. Using quality modeling techniques at the design stages of development can help to identify faults early on, so aiding later maintenance; in later sections, some of these techniques are described. In developing a quality model, we use maintainability as an example, and show how to develop a quality model which evaluates its components. In any computer system, the problem of complexity is always going to be a consideration. The different types of complexity cover a wide area from understanding the nature of the initial problem to understanding the developed system; the problem of maintaining systems can be seen as a problem of overcoming the complexity of understanding the system and its continued operation.

Relatively new paradigms such as the object-oriented style of design and programming offer new challenges in the assessment of complexity, and necessitates new ways of measuring that software. Metrics proposed in the past for the procedural paradigm are not always relevant or applicable. In the next section, we consider the assessment of complexity in systems, first, inherent to a particular system, where we analyze a system in isolation and the features which contribute to its complexity, and, second, between different systems, to which we give the term comparative complexity. A feature of software which affects both inherent and comparative complexity is that of the coupling of modules. By coupling, we

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

mean the measure of the strength of association between one module and any other module with which it interacts. Many metrics have been proposed which purport to capture the extent of coupling in systems, but it is still an area of current interest and debate, as is software cohesion.

A particular area of importance in the software metrics world is that of software tools to collect the essential data for analysis. Software tools automate time-consuming data collection tasks and are becoming both more prevalent and sophisticated in what they can achieve. However, they are still crude in terms of the time and resources they consume (often prohibitive), the types of systems which they can be applied to (they are often language dependent), and their reliability (frequent failures in the software tool mean wasted time and resources). In the final section, we consider the availability of tools, the potential for more powerful tools which may overcome some of the problems currently faced (the size and complexity of systems), and how they might meet future requirements and trends in the area of software engineering.

MOTIVATION FOR USING SOFTWARE METRICS

Measurement in everyday life is invaluable in understanding the entities around us and the important attributes of those entities; the same is true of software metrics in understanding software and its characteristics. In the 1970s, metrics tended to focus on features of the product under examination. McCabe's cyclomatic complexity metric is a good example of one of these early measures (3). This metric is obtained by analyzing the control flow through a piece of software; an optimum level of the metric may be prescribed, beyond which a program is considered overly complex. The complexity metric has also been used as the basis of prediction systems for estimating (among other things) testing difficulty and maintenance difficulty.

In the 1980s, the focus shifted to the analysis of models for cost estimation and measurement of program specifications. The best example of this type of model is COCOMO, the constructive cost model (4), a cost model derived from analysis of sixty-three software projects. The model can be tailored to the requirements of the particular project, and has recently been updated to take new forms of software development into account (5). The early 1990s saw a further shift toward analyzing the development process, and measuring features of the development process. For example, how good is the design in terms of maintenance needs during the development cycle? What is the relationship between the faults discovered in a system and the types of structures used in the design? This has particular significance in the object-oriented paradigm, where we would want to know the effects of using a deep (as opposed to a shallow) inheritance hierarchy, where a deep inheritance hierarchy implies a system with a large amount of inheritance.

The Role of Measurement Theory

The use of sound measurement theory principles, applicable to measuring real-world entities, is a prerequisite for the design of any metric. However, there has been a lack of attention paid to such rigor in the metrics field in the past. For example, a metric must use the appropriate measurement scale. A counting metric would be measured on an absolute scale, an example of a counting metric being the number of lines of code. An ordered metric, which expresses a range of values such as preference, would be measured on an ordinal scale. In fact, a metric can be identified as belonging to one of five major scale types (nominal, ordinal, interval, ratio, or absolute). The representation condition of measurement (RC) (1) emphasizes the importance of an empirical (observed) relationship also holding true in the mathematical world. For example, if a program X appears to be longer than program Y, and assuming that the number of lines of code is a (proposed) measure of program length, then we would expect the number of lines of code for X to be larger than the number for Y. According to Fenton (1), measurement is important for three basic activities: to understand, to control, and to improve. If measures are available to help understand what is happening during development, the added control allows an easier assessment of future courses of direction, and better prediction of likely outcomes; this applies as much to the maintenance process as any. The intended user of a metric must be borne in mind when choosing a metric, since software developers, testers, middle management, etc., have different data requirements. The scope of software metrics extends from cost and effort estimation to assessment of reliability in software. The notion of control is central to all these areas. According to De Marco (6), "You cannot control what you cannot measure." Fenton (1) adds to this, "You can neither predict nor control what you cannot measure."

These two quotes imply that we must also be realistic in choosing attributes of software which can be measured. There is no value in believing one has measured one attribute of software to find one has measured a different attribute. A distinction should be made between *internal* directly measurable attributes such as size, and *external* indirectly measurable attributes such as comprehensibility, maintainability, etc. Metrics are usually based on internal (low-level) attributes, derived from external (high-level) attributes.

The Role of Quality Models

Use of a proper quality modeling technique helps to identify the appropriate external and internal attributes. For example, the goal question metric (GQM) (7) or factor criteria metric (FCM) (8) model establish the high-level attributes, such as reliability and productivity to be measured and how these are decomposed to the low-level capturable metrics. Using a model which is rigorous in its application also means there is a better chance that the model can be used effectively for prediction purposes.

Application of quality models and the metrics they give rise to allows the following sorts of questions to be answered:

1. Is the quality of the software we produce improving? Here, quality might be expressed in terms of the size of the maintenance task, potential metrics for which might be: mean time to failure (MTTF), measuring the expected time between one failure and the next, mean time to repair (MTTR), measuring the expected time taken to fix a bug, number of modification requests, etc.
2. Is our productivity improving? Here, productivity might be expressed in terms of costs of personnel. Characterisation of any entities and attributes we are interested in measuring can be achieved by identifying the prod-

ucts, processes, and resources used during software development; this is often referred to as a *metrics framework*.
3. Are we meeting costs and time deadlines?
4. Are our goals being met? According to Gilb (9), "Projects without clear goals will not achieve their goals clearly."

It is in these areas where measurement and the application of metrics have the most to offer. The main difficulties in software development are characterized by development which overruns both budget and time scale, and by high maintenance costs. Measurement has an important role to play in alleviating these problems through control, understanding, and improvement.

The Need for Rigor

Measurement as a software engineering discipline has attracted considerable interest over the years. The lack of application of sound measurement theory to software and the lack of empirical studies using that theory as a foundation might explain the controversy surrounding software metrics in the past. More recently, progress has been made toward laying solid theoretical foundations for the application of measurement theory to software engineering practice (10,11). Ensuring consistency in the application of software metrics is essential for their widespread adoption. At present, however, there is still much debate in the metrics community on the correct framework to adopt for software measurement validation (12).

In the next section, we turn to recent work in these areas, and the potential this work has for the development of quality software. We consider the issues of theoretical validation and empirical evaluation.

CONSISTENT APPLICATION

A major problem with introducing any potential software metric is ensuring that it conforms to a validation process general enough to allow any metric to be assessed, yet rigorous enough to impose strict conditions on what constitutes a valid metric. A major consideration for any proposed metric is to ensure that it is theoretically valid, and conforms to strict mathematical properties.

Theoretical Validation

In measuring attributes of software, Fenton (1) poses several questions, applicable for real-world entities, but more difficult to answer when considering software. The questions posed highlight the more abstract nature of software, and the problems associated with software measurement.

1. How much must we know about an attribute before it is reasonable to consider measuring it?
2. How do we know if we have really measured the attribute we wanted to measure?
3. What meaningful statements can we make about an attribute and the entities that possess it?
4. What meaningful operations can we perform on measures?

The theoretical approach to the validation of metrics requires us to clarify what attributes of software we are measuring, and how we go about measuring those attributes (10,13,14); a metric must measure what it purports to measure.

Fenton (1) describes the *representation condition* (the basis of the representational theory of measurement), satisfaction of which is the prerequisite for any metric to be viewed as valid. The representation condition states that any measurement mapping must map entities into numbers, and empirical relations into numerical relations, such that those relations are preserved. In other words, our observations in the real world must be reflected in the numerical values we obtain from the mathematical world.

Briand and Morasca (11) argue for the need to unambiguously define the important features of software through a sound mathematical framework, not specific to any particular feature of software, yet rigorous in its application being based on specific mathematical concepts. As an example from this work, and a theme which we will return to later, consider the complexity of a system expressed in terms of the level of coupling in that system. Here, coupling refers to intermodule relationships. Valid coupling metrics should be based on mathematically proven laws. For example, merging two modules (between which there was no previous relationship) should not cause the level of coupling in the system to rise. Equally, merging two modules (between which there was previously a strong relationship) should cause the level of coupling in the system to fall. Other properties such as nonnegativity are also suggested: a system cannot have a negative size and cannot have a negative level of coupling.

In Fenton and Pfleeger (1), correct selection of the scale type and the potential impact this has for statistical operations on collected data are emphasized. Often, an incorrect statistical analysis can be performed following the wrong choice of scale type. The difficulty is knowing which scale type is appropriate, a question that is far from straightforward. In the case of two sets of data measured on two different scale types, appropriate transformations can be made to preserve the integrity of the statistical operations on the two sets of data.

Kitchenham et al. describe a list of features of metrics which must hold for that metric to be valid (10). For all metrics, the following criteria must hold for a metric to be considered valid:

- For an attribute to be measurable, it must allow different entities to be distinguished from one another.
- A valid measure must obey the representation condition, that is, it must preserve all intuitive notions about the attribute and the way in which the measure distinguishes between entities.
- Each unit of an attribute contributing to a valid measure is equivalent.
- Different entities can have the same attribute value within the limits of measurement error.

For an indirect metric, the following criteria must also be considered. The metric should:

- Be based on an explicitly defined model of the relationship between attributes.

- Be dimensionally consistent.
- Not exhibit any unexpected discontinuities.
- Use the appropriate measurement scale.

Validation Example

Take as an example, in the object-oriented paradigm, the weighted methods per class (WMC) metric of Chidamber and Kemerer (16). The metric was proposed as a measure of the complexity of a class, the hypothesis being that the more methods there are in a class, the more complex that class. Each method of a class may perform a particular function on an object. So, for example, a class *BankAccount* would have methods to return the account number, to return the balance, to handle deposits, to handle withdrawals, and so on. To validate WMC against the first four criteria, we must decide what the WMC measures directly; WMC is a direct (countable) measure of the size of a class.

Turning to the first set of criteria, it can be seen how one class may be distinguished from another class in terms of the WMC value, if, for example, it has a different number of methods. It is straightforward to see that the representation condition holds, since, under normal conditions, the greater the number of methods in a class, the greater the size of that class. Each unit contributing to the metric is viewed as equivalent (since we make no distinction between one method and another), and different classes can have the same number of methods. We can therefore view the WMC metric as a valid indicator of the size of a class.

To validate WMC against the second set of criteria, we must decide what WMC indirectly measures, and see if a relationship exists between that direct measure (size) and the indirect measure. Consider WMC as an indirect measure of maintainability (16). Maintainability is a difficult concept to define, but for the sake of argument assume it to mean modifiability (i.e., how easy it is to make a modification to a class). The question then is: Are small classes easier to modify than large classes?

This question can only be answered in particular circumstances, after the collection of a large amount of empirical evidence. There are other hypotheses which we would also like to test, such as the relationship between WMC and development effort. Through experimentation we can begin to understand more about the validity of WMC. In addition to being theoretically valid, a metric should be shown to be valid in an empirical sense. For real, large-scale systems, we would like to know if WMC gives a good indication of the effort to develop a class. This hypothesis forms the basis of an experiment, carried out by collecting data from real systems, and identifying whether a significant relationship exists between the WMC values for a system and a metric such as development time; these activities form what is known as an empirical evaluation (10,13,14,17).

Empirical Evaluation of Metrics

It is preferable to have a metric which is both theoretically valid and can be shown to be of use through empirical evaluation. For example, an empirical investigation has been described which attempts to identify design metrics most useful to the software engineer (18). A metric based on information flow is introduced, and found to correlate highly with development effort. In this study, various code metrics are shown to be poor indicators of development effort.

Design metrics based on information flow were described and evaluated using data from a communications system (19). The ability of the design metrics to identify change-prone, error-prone, and complex programs was compared with that of simple code metrics, that is, lines of code, number of branches and so on. The conclusion arrived at was that code metrics were better at identifying such programs than the design metrics. The Chidamber and Kemerer set of object-oriented metrics were validated empirically using eight medium-sized information management systems, and were assessed as predictors of fault-prone classes (13). For the WMC metric in particular, the experimental hypothesis to be statistically tested was: A class with significantly more member functions than its peers is more complex and, therefore, tends to be more fault-prone.

Fault data from the eight systems were then collected, and relationships identified. In this case, the WMC hypothesis was supported, leading to the conclusion that larger classes are indeed more fault-prone.

EMPIRICAL SOFTWARE ENGINEERING

Empirical software engineering can be defined as: The study of software-related artifacts for the purpose of characterization, understanding, evaluation, prediction, control, management, or improvement through qualitative or quantitative analysis. The software research crisis has arisen from the continued practice of advocacy research at the expense of all other possible forms of research (20). It is suggested that research should be broken down into four phases:

1. The informational phase. Gather or aggregate information via reflection, literature survey, people/organizational survey, or poll.
2. The propositional phase. Propose and/or build a hypothesis, method or algorithm, model theory, or solution.
3. The analytical phase. Analyze and explore a proposal, leading to a demonstration or the formulation of a principle or theory.
4. The evaluative phase. Evaluate a proposal or analytic finding by means of experimentation (controlled) or observation (uncontrolled, such as a case study) perhaps leading to a substantial model, principle, or theory.

Fenton (1) recommends examining the following criteria when considering the viability of an experiment:

1. Is it based on empirical evaluation and data rather than intuition and advocacy?
2. Does it have a good experimental design? There are many possible ways an experiment can be designed.
3. Is it a toy situation or a real situation?
4. Are the measurements appropriate to the goals of the experiment?
5. Was the experiment run long enough to evaluate the true effects of the change in practice?

Empirical evaluation usually begins with questions such as:

Does the use of formal methods in systems development influence the quality of the end product?

Does the level of training given to development staff affect the final reliability of software?

A distinction is made between independent variables and dependent variables. In the last example, the independent variable would be the level of training, and the dependent variable the reliability of software. A number of examples will help to illustrate the sort of experimentation currently being undertaken, and the objectives which such experimentation seeks to fulfill. A good introduction to the area of experimentation, and the steps in setting up and carrying out an experiment can be found in Refs. 21 and 22. The following section briefly reviews some recent results obtained through empirical software engineering research.

Examples

An investigation into the effects of using formal methods to develop an air-traffic control information system examined a number of methodologies during development, ranging from Milner's calculus of communicating sequential processes to a totally informal approach (23). For each approach, metrics such as total lines of delivered code, number of changes to delivered code, percentage of modules changed, and number of faults discovered were collected. Results suggested that using an approach based on formal methods led to code which was relatively simple and easy to test.

A recent paper describes an experiment in which the maintainability of object-oriented design documents was compared with that of structured design documents (24). The conclusions arrived at were that there was no evidence to suggest that object-oriented design documents were easier to maintain than structured design documents, and if anything, object-oriented design documents are more sensitive to poor design.

Roper et al. (25) describe an empirical study carried out to compare three defect detection techniques: code reading, functional testing, and structural testing using branch coverage. The conclusions were that individual techniques were broadly similar in terms of observing failures and finding faults, but used in combination are much more effective. Porter et al. (26) describe the results of an experiment to detect faults in requirements documents. They introduce a scenario-based approach which improves on existing ad hoc and checklist detection techniques. The scenario method (a more specific version of the checklist) was shown to identify more faults.

The findings of an experiment in which the benefits to maintenance of using modular code against nonmodular (monolithic) code were examined (27). The experiment produced results which went against the traditional view that modular code was easier to maintain than nonmodular code; it was suggested that the initial experimental technique used was at fault. This highlights the problem of placing too much faith on the results of one experiment.

Cartwright and Shepperd (28) describe an empirical investigation of an industrial object-oriented system. Metrics were collected from a large telecommunications system. The two main results were the small use of inheritance in the system (133,000 lines of code), and the ability to predict numbers of defects based on simple counts rather than complex suites of metrics.

A study of two releases of a major commercial system revealed a counterintuitive relationship between pre- and post-release faults (29). Modules among the most fault-prone prerelease were found to be among the least fault-prone postrelease. Conversely, modules among the most fault-prone postrelease were found to be the least fault-prone prerelease; the need for further empirical investigations to support or refute these claims is emphasised.

Threats to Validity

In performing an experiment, three types of threats to the validity of that experiment have to be considered. These are *external, internal,* and *construct* validity threats; each is now explained.

External validity is the degree to which results from the experiment can be generalised to the population, that is, other groups. The use of students as subjects in an experiment may be a threat to external validity, for example.

Internal validity is the degree to which we can conclude that the dependent variable is accounted for by the independent variable. For example, to what extent can we say that the reliability of software is due to the level of training received by development staff? Are there other variables which should be considered?

Construct validity is the degree to which the independent and dependent variables accurately measure what they purport to measure. For example, are we really measuring the reliability of software or something completely different?

Many of the empirical investigations described seek to characterize a particular feature of software. For example, an investigation of a metric for maintainability may be based on the relationship between the metric and the number of faults found in a system. The main problem is the lack of studies on large systems; it is difficult to determine whether or not results from small systems will scale-up.

Benefits from Empirical Analyses

Until now, little empirical analysis has been performed using a proper scientific basis (20,30). Much of the evidence put forward to support claims about the way software should be engineered has been anecdotal, with no study of real systems taking place. An example of such a claim might be: "By using a particular development technique, maintenance can be reduced by 30%."

It is only by analyzing large numbers of systems empirically and building up a base of experience that such claims can be justified. The Software Engineering Laboratory (SEL) at the University of Maryland (in conjunction with NASA) has, as a specific goal, the improvement of the software process through an experience factory (31). Lessons from analyses of previous projects are stored for future projects. Two requirements must be met for a rigorous, scientific foundation to software engineering:

1. A top-down evolutionary framework in which research can be focused and logically integrated to produce mod-

els which can then be evaluated and tailored to the application environment.

2. A laboratory associated with the software artifact that is being produced and studied to develop and refine comprehensive models based upon measurement and evaluation.

Greater understanding of the way software behaves and ways of improving the development of software are therefore two of the major benefits of empirical analyses. It is only through understanding that control can be exercised and improvements made. The need to understand the nature of software is particularly relevant to the area of maintainability, which accounts for a large percentage of the development budget. In the following section we discuss how maintainability can be characterised and measured at an appropriate level.

SOFTWARE SYSTEM MAINTAINABILITY EVALUATION

Maintaining software in the sense of ensuring it continues to provide the functionality it is supposed to is a time-consuming and difficult task. In the following sections, maintenance refers to all types of maintenance: corrective maintenance needed to correct errors in the software, adaptive maintenance needed to preserve the purpose of the program in a changing environment, preventive maintenance in anticipation of changes, and perfective maintenance to improve a program in terms of its functionality. Care must also be taken to distinguish between *faults* and *failures.* A failure is defined as a manifestation of a fault. Since all faults do not necessarily lead to failures it is likely that a system will contain undiscovered faults throughout its lifetime. This has serious implications for maintenance and the effort that should be invested in the testing process. Conversely, many faults discovered during testing and reviewing may be benign in the sense that they would never realistically trigger a failure in operation. Maintenance encapsulates many areas of system behavior. In the object-oriented paradigm, concepts such as inheritance and polymorphism present difficulties by their very nature. Consider the problem of having to make a modification to a class at the root of an inheritance hierarchy (from which all other classes inherit). All inheriting subclasses are potentially affected. As such, inheritance can be viewed as a form of coupling, in the sense that changes in one part of a system may have serious implications for other parts of the system.

Maintainability is related to system stability. Brand et al. (32) suggest developing separate metrics for classes in an OO system which are stable, that is, that are not likely to be changed (e.g., library classes), and those which are unstable and more likely to be changed. The overriding point to note is that maintainability is an important factor in the assessment of software quality. Exactly how it influences quality can be better understood by developing a quality model in which its constituent parts are evaluated.

A Quality Model

Many models have been suggested as a means of identifying quality. Most notable amongst these are Boehm's quality model for cost and effort estimation and the factor criteria metric (FCM) model (in which the quality factors are broken down into the respective criteria, from which metrics can be developed). There are also quality guidelines based on the ISO 9126 standard (33) and the QMS subsystem (34), which identify the individual quality factors making up the overall view of quality. For example, reliability has related concepts of availability, correctness, and fault-tolerance; these criteria can be broken down further and the metrics formed at the lowest level of the treelike structure produced. So, for example, availability can be broken down into directly measurable attributes such as percentage of time a machine is available over a specific time period, response times, and maximum loads.

The GQM Approach. The GQM approach (7) can be used in a variety of settings. It can be used to control a software project by monitoring progress and allowing feedback, to analyze a current scenario with a view to its improvement, or as in this article, to determine in some detail the characteristics of a feature of software quality.

The GQM approach defines a quality model on three levels:

- Conceptual level (goal). A goal is defined with respect to various models of quality, from various points of view, and relative to a particular environment.
- Operational level (question). A set of questions is used to define aspects of the object and to characterise the goal in more detail.
- Quantitative level (metric). One or more metrics are associated with every question.

Approach Adopted

A hybrid approach to the evaluation of maintainability is adopted here. It draws on parts of the GQM approach, the ISO 9126 standard, and QMS subsystem. The overall goal (in keeping with the GQM approach) is: To improve the prediction of quality from the developer's viewpoint.

The question then has to be asked: What are the individual factors contributing toward the overall view of quality? These can then be listed, among which is the maintainability factor. Other factors might include reliability, usability, functionality, and reusability. It is open to the developers of the model to decide exactly which factors are most important in their particular case. For a safety-critical system, the reliability factor will be more important than a factor such as reusability. From the ISO model, maintainability is defined as the set of attributes that bear on the effort to make specified modifications. From the QMS model, maintainability is defined as the ease with which faults can be diagnosed. The next question which has to be asked is: What are the related concepts for maintainability?

Related concepts can then be decided on; these could be:

- Understandability. How easy is it to understand a system and its components?
- Testability. How easy is it to test a system and its components?
- Stability. How stable is a system in terms of the stability of its components?
- Modifiability. How easy is it to modify components of the system?

These then are the candidate concepts for maintainability. The same process could be carried out for the other four factors. The next question which needs to be asked is: Which of the last concepts are most important when considering maintainability (bearing in mind the original goal)?

We might decide that understandability and modifiability are the key factors here. From a developer's viewpoint, these two factors are very likely to be a major consideration. The next question which needs to be asked is then: What are the constituents of each of the concepts: modifiability and understandability?

This is the stage at which the metrics start to evolve from the higher level attributes via questions such as: Which process metrics and which product metrics are most appropriate for each of the chosen factors (modifiability and understandability)?

For modifiability we might arrive at the following process metrics:

- Number of modification requests (where a modification request is a request for change other than a fault fix).
- Time taken for a modification request to be completed.
- Number of modification requests/KLOC (KLOC refers to thousands of lines of code).
- Number of faults.
- Time taken for a fault to be fixed.
- Number of faults/KLOC.

Measuring maintainability is a matter of capturing features of the maintenance process. Product metrics for modifiability would be measures which are likely to be related to the chosen process metrics. Some candidate product metrics might be:

- Number of modules in the system. In the object-oriented paradigm, this might be the number of classes or methods.
- Number of variables per module. In an object-oriented paradigm, this might be number of attributes per class.
- Number of variables in the system.
- Number of lines of code.
- Number of relationships between modules (as a measure of coupling). This could be expressed in terms of calls by modules to other modules.

In addition, there may also be measures specific to the paradigm under examination: for example, measures of the inheritance structure, and types of relationships between classes, both of which may affect the modifiability of a system.

For understandability, we would consider the following process metrics:

- Time taken to diagnose a fault.
- Time taken to understand requirements for a modification prior to coding.
- Average time taken to diagnose a fault.

The product metrics could be similar to those for modifiability.

Once the process and product metrics have been decided on, it is a question of identifying relationships between the product and process metrics. For example, is there a relationship between the time taken to diagnose a fault, and the level of inheritance in a system? Does the number of variables in a module affect the time taken to understand the requirements for a modification? Each of these questions could form a hypothesis as follows: It is more difficult to diagnose a fault in a system with a deep inheritance hierarchy than in a system with little or no inheritance. It takes longer to understand the requirements for a modification if a module has a relatively large number of variables.

These hypotheses can then be tested empirically, and in this way an understanding of the features of the system under examination can be obtained. From a high-level view of what constitutes quality, a set of hypotheses have been developed which can be tested on real systems. Figure 1 illustrates the stages by which the quality model was developed. Decomposing the concept of quality into its constituent parts until the low-level metrics are produced is an example of the divide and conquer approach to problem solving. The purpose of performing empirical analyses is to try and understand the complex nature of software, and to seek ways of improving software development. In the next section, the concept of software complexity is examined.

Goal:	To improve the prediction of quality from the developer's viewpoint
Question:	What are the individual factors contributing toward the overall view of quality?
Answer:	Maintainability, Reliability, Usability, Functionality, Reusability
Question:	What are the related concepts for maintainability?
Answer:	Understandability, Testability, Stability, Modifiability
Question:	Which of the last concepts are most important when considering maintainability (bearing in mind the original goal)?
Answer:	Modifiability, Understandability
Question:	What are the constituents of each of the concepts: modifiability and understandability?
Question:	Which process metrics and which product metrics are most appropriate for each of the chosen factors (modifiability and understandability)?
Metrics:	Modifiability process metrics: Number of modification requests, Time taken for a modification request to be completed .... Understandability process metrics: Time taken to diagnose a fault, ...

Figure 1. Developing a GQM quality model.

A VIEW OF COMPLEXITY

In recent years there has been a tendency to view complexity as relating to products of the development process, expressed in terms of the size or functionality of a system. A classic example is that of McCabe's complexity metric which measures the control flow through a program (3).

In fact, complexity *per se* is by no means an easy concept to define. Many attempts have been made to identify complexity within software (1,5,35,36). Complexity is sometimes assumed to be synonymous with size or functionality, in the belief that larger systems are more complex. Lorenz and Kidd (35) define complexity loosely as that characteristic of software that requires effort (resources) to design, understand, or code; as such, complexity covers all stages of the development cycle. In the next section, complexity is examined according to a framework discussed by Fenton (1).

Inherent Complexity

Complexity can be defined as falling into one of four main categories. Any system will contain these four categories of complexity; as such, they can be seen as the inherent features of a system's complexity.

Problem Complexity. This measures the complexity of the underlying problem. The complexity of a problem is the amount of resources required for an optimal solution to the problem, and complexity of a solution as the resources needed to implement a particular solution. For some problems, it can be shown that there is no solution, and hence, it cannot be implemented by a computer. Solutions to other classes of problems can consume too many machine resources making them prohibitively expensive and time-consuming to implement.

Algorithmic Complexity. This reflects the complexity of the algorithm implemented to solve the problem. This is, in effect, a measure of the efficiency of the software produced. Efficiency may be seen as becoming less of a consideration as hardware becomes cheaper and faster.

Structural Complexity. This measures the structure of the software used to implement the algorithm. To obtain this measure, we might look at control flow structure, hierarchical structure, and modular structure, for example.

Cognitive Complexity. This measures the effort required to understand the software.

Problem complexity and algorithmic complexity relate strongly to the establishment of initial user requirements and the alternatives for providing a solution to the problem. An important area of research at present centers on verifying requirements to ensure that they have been captured correctly. This is essential in view of a recent study by Schneider et al. (37), where it is claimed that finding and fixing a software fault after delivery is one hundred times more expensive than finding it during the requirements and early design phase. Thus, problem and algorithmic complexity are usually factors encountered during the early design stages of development.

Structured code metrics such as McCabe's complexity measure can be used to analyze the structural complexity of software. Early attempts to measure complexity were based on the lexical content of programs; for example, Halstead's metric (38) counted the number of operators and operands in a program. As another example, the complexity metrics of Yin and Winchester (39) assessed the complexity of a system using the module calling structures, represented in the form of graphs. Such analysis gives a good indication of any structural weaknesses, and the dependancies between modules. The complexity metric (40) for the complexity of a procedure was defined as:

$$\text{length} \times (\text{fan-in} \times \text{fan-out})^2$$

Here, the length of a procedure represents number of lines of source text. Fan-in of a procedure is the number of flows of information or control into a procedure and fan-out is the number of flows of information or control out of a procedure. Similarly, the IF4 complexity metric suggested by Shepperd (36), and based on similar work (40) identifies two types of information flow. Local information flow is based on explicit communication between modules; global information flow is based on access to shared data structures. A module call with an argument would be an example of local information flow. An example of global information flow occurs if one module updates a data structure and another module reads from that data structure.

Analysis of software using these types of metrics can be useful for understanding code structure. Henry and Kafura (40) describe such metrics as useful management aids, important design tools, and important in establishing a foundation for comparing language constructs and methodologies. However, a growing trend in the metrics community is to look to collect metrics as early as possible in the development process for the purpose of prediction, rather than using structural complexity metrics taken from code.

Cognitive complexity is clearly subjective and consequently difficult to measure accurately. Boehm proposes a subjective understanding scale for COCOMO 2.0 (5), which ranks software according to structure, application clarity, and self-descriptiveness. Software understanding is rated on an ordinal scale of 1 to 5. So, a program with strong modularity, where there is a clear match between program and application world-views and which is documented well would be considered easily understandable, and rated accordingly. Alternatively, a program with low cohesion with no match between program and application world-views, and containing obscure code would be rated at the other end of the scale.

Comparative Complexity

In determining the comparative complexity of systems, a different set of problems arise. There are clearly a large number of variables which affect all forms of complexity, not least of which is the size of the system. As far as cognitive complexity is concerned, we also need to consider the programming languages and environments, developer's experience, etc. Clearly, a manager needs to baseline project complexities over a number of years to obtain comparative complexity profiles.

Unfortunately, there is a dearth of empirical investigation using industrial-sized systems. This makes it even more difficult to draw conclusions from past empirical analyses largely carried out on small-scale manageable systems; it is difficult to judge the comparative complexity of large systems based on such investigations. Collections from industrial-sized systems in differing application domains should be made. This would provide design guidelines for the different types of system (as per COCOMO); the field of software metrics is only just beginning to address some of these problems. In the next section, the question of the availability of tools to carry out these collections is addressed.

MEASURING TOOLS

An important part of the measurement process is having the tools to collect metrics automatically. We would like to be able to apply such tools to both large and small systems, and to systems in different application domains. Automated tools do exist to support the collection of metrics but are only as useful as the metrics suites themselves. Manual collection of data can be a time-consuming, error-prone, and cumbersome process. The need to carry out manual data collection arises from the fact that many of the metrics suites currently available have no supporting tools for automated data collection. Manual data collection is time-consuming and error-prone. However, automated data collection using toolkits also suffers from various problems:

- It can also be a time-consuming process, requiring a large amount of machine resources.
- Few automated tools exist to capture metrics from design documents.

Consequently, there is an urgent need for new tools to help in the collection of metrics. Such tools should capture metrics at the right level of abstraction, be applicable to systems of all sizes and application domains, be reliable, and not consume overly large amounts of machine resources. The availability of supporting tools is also crucial to a proper empirical evaluation. For example, as well as tools which collect metrics from the design, tools for collecting process metrics such as development times and testing times should also be available. Time should also be invested in producing planning and forecasting tools to aid the evaluation of the metrics concerned.

There is a clear need to speed up the learning process in terms of which metrics are useful, and which are not. This can only come through provision of the required data through efficient, automated measuring tools relevant to the state of the art in the metrics field.

CONCLUSIONS AND FUTURE TRENDS

In this article we have described the current state of the art in the field of software metrics. Software engineering is an intrinsically difficult and multidisciplinary profession which plays an increasingly important part in the lives of everyone. Software metrics contribute to software engineering by offering a way of understanding the critical features of software, and as such, have an important role to play in controlling the software development process. The work on theoretical validation and empirical evaluation of metrics is still in its formative stages, but has already added rigour to an area which previously had none. A great deal of work still needs to be done in both areas, particularly in empirically evaluating metrics. Development of comprehensive quality models helps us to identify the most critical attributes of software particular to a system and ensures that the metrics collected are both relevant to the attributes being examined and useful. The combination of rigor and well-defined models of quality help to capture features such as the maintainability of systems. Handling the complexity inherent in all systems then becomes easier. More sophisticated tools which can cope with large scale systems easily and efficiently will be needed in the future to facilitate accurate and timely data collection.

It is the combination of theoretical validation, empirical evaluation, and practical application of supporting tools which will enable some of the current problems faced in software engineering to be tackled.

ACKNOWLEDGMENT

This work is supported by UK EPSRC project GR/K83021.

BIBLIOGRAPHY

1. N. E. Fenton and S. L. Pfleeger, *Software Metrics, A Rigorous and Practical Approach,* International Thomson Computer Press, 1996.
2. B. A. Kitchenham and S. L. Pfleeger, Software quality: The elusive target, *IEEE Software,* 12–21, 1996.
3. T. McCabe, A complexity measure, *IEEE Trans. Softw. Eng.,* **2**: 308–320, 1976.
4. B. W. Boehm, *Software Engineering Economics,* Englewood Cliffs, NJ: Prentice Hall, 1981.
5. B. W. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, COCOMO 2.0, *Annals Software Eng.,* **1**: 1–24, 1995.
6. T. De Marco, *Controlling software projects: Managements, measurement and estimation,* Yourdon Press, 1982.
7. V. R. Basili and H. D. Rombach, The tame project: Towards improvement-oriented software environments, *IEEE Trans. Softw. Eng.,* **14**: 758–772, 1988.
8. J. A. McCall, P. K. Richards, and G. F. Walter, Factors in software quality, Tech. Rep. NTIS AD/A-049 014,015,055, US Rome Air Development Center, 1977.
9. T. Gilb, *Software Metrics,* London: Winthrop Publishers, 1977.
10. B. A. Kitchenham, S. L. Pfleeger, and N. Fenton, Towards a framework for software measurement validation, *IEEE Trans. Softw. Eng.,* **21**: 929–944, 1995.
11. L. C. Briand and S. Morasca, Property-based software engineering measurement, *IEEE Trans. Softw. Eng.,* **22**: 68–85, 1996.
12. S. Morasca, L. C. Briand, E. J. Weyuker, and M. V. Zelkowitz, Comments on: Towards a framework for software measurement validation, *IEEE Trans. Softw. Eng.,* **23**: 187–188, 1997.
13. V. R. Basili, L. C. Briand, and W. L. Melo, A validation of object-oriented design metrics as quality indicators, *IEEE Trans. Softw. Eng.,* **22**: 751–761, 1996.
14. N. E. Fenton, Software measurement: A necessary scientific basis, *IEEE Trans. Softw. Eng.,* **20**: 199–206, 1994.
15. L. C. Briand, K. El Emam, and S. Morasca, On the application of measurement theory in software engineering, *Emp. Software Eng. J.,* **1**: 61–88, 1996.
16. S. R. Chidamber and C. F. Kemerer, "Moose: Metrics for object oriented software engineering," in *Workshop on Processes and Metrics for Object Oriented Software Development, OOPSLA '93, Washington,* 1993.

17. N. F. Schneidewind, Methodology for validating software metrics, *IEEE Trans. Softw. Eng.,* **18**: 410–422, 1992.

18. D. C. Ince and M. J. Shepperd, An empirical and theoretical analysis of an information flow based design metric, in *Proceedings of the European Software Engineering Conference,* Warwick, UK, pp. 11–16, 1989.

19. B. A. Kitchenham, L. M. Pickard, and S. J. Linkman, An evaluation of some design metrics, *Software Eng. J.,* 50–58, Jan 1990.

20. R. L. Glass, The software research crisis, *IEEE Software,* 42–47, November 1994.

21. S. L. Pfleeger, Design and analysis in software engineering: Part 1: The language of case studies and formal experiments, *ACM Sigsoft Software Eng. Notes,* **19** (4): 16–20, 1995.

22. S. L. Pfleeger, Experimental design and analysis in software engineering: Part 2: How to set up an experiment, *ACM Sigsoft Software Eng. Notes,* **20** (1): 22–26, 1995.

23. S. L. Pfleeger and L. Hatton, Investigating the influence of formal methods, *IEEE Computer,* 33–43, 1997.

24. L. Briand, L. Bunse, J. Daly, and C. Differding, An experimental comparison of the maintainability of object-oriented and structured design documents, in *Proceedings of Empirical Assessment in Software Engineering (EASE) '97, Keele, UK,* 1997.

25. M. Roper, et al. Comparing and combining software defect detection techniques: A replicated empirical study, in *Proceedings of Empirical Assessment in Software Engineering (EASE) '97,* Keele, UK, 1997.

26. A. A. Porter, L. G. Votta, and V. R. Basili, Comparing detection methods for software requirements inspections: A replicated experiment, *IEEE Trans. Softw. Eng.,* **21**: 563–575, 1995.

27. J. Daly, et al, Verification of results in software maintenance through external replication, in Proceedings of the *IEEE International Conference on Software Maintenance (ICSM '94),* 1994.

28. M. Cartwright and M. Shepperd, An empirical analysis of object-oriented software in industry, in *Bournemouth Metrics Workshop,* Bournemouth, UK, April 1996.

29. N. E. Fenton and N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, *submitted for publication,* 1997.

30. N. E. Fenton, S. L. Pfleeger, and R. L. Glass, Science and substance: A challenge to software engineers, *IEEE Software,* 86–95, 1994.

31. V. R. Basili, et al, The software engineering laboratory: An operational software experience factory, in Proceedings of the International Conference on Software Engineering, pp. 370–381, 1992.

32. L. C. Briand, J. W. Daly, and J. Wust, "A unified framework for coupling measurement in object-oriented systems," isern-96-14, Fraunhofer Institute for Experimental Software Engineering, 1996.

33. ISO/IEC, Joint technical committee: Information technology—software product evaluation—quality characteristics and guidelines for their use, international standard, ISO/IEC, 1991.

34. B. A. Kitchenham, J. D. Walker, and I. Domville, Test specification and quality management: Design of a qms sub-system for quality requirements specification, Project Deliverable A27, Alvey Project SE/031, Nov 1986.

35. M. Lorenz and J. Kidd, *Object-oriented Software Metrics,* Englewood Cliffs, NJ: Prentice Hall Object-Oriented Series, 1994.

36. M. J. Shepperd, Design metrics: An empirical analysis, *Software Eng. J.,* **5** (1): 3–10, 1990.

37. G. M. Schneider, et al, An experimental study of fault detection in user requirements documents, *ACM Trans. Software Eng. Method.,* **1** (2): 188–204, 1992.

38. M. H. Halstead, *Elements of Software Science,* New York: Elsevier, 1977.

39. B. H. Yin and J. W. Winchester, The establishment and use of measures to evaluate the quality of software designs, in *Proceedings of the ACM Software Quality Workshop,* pp. 510–518, ACM, 1978.

40. S. Henry and D. Kafura, Software metrics based on information flow, *IEEE Trans. Softw. Eng.,* **7**, 510–518, 1981.

R. Harrison
S. Counsell
University of Southampton