CHAPTER 4

Fundamental Issues in Software Measurement

James M. Bieman, Norman Fenton, David A. Gustafson,
Austin Melton, and Linda M. Ott

4.1 Introduction

This chapter builds on the previous one. In the previous chapter the emphasis is on measurement theory, and software measurement is discussed as a type of measurement. In this chapter the emphasis is on software measurement, and ideas from measurement theory are used to guide and direct our development of software measurement ideas and methods.

4.1.1 Terminology

Interestingly, the term *metric* has more than one meaning in software measurement. It is used in at least the three ways depicted below:

1. A number derived from a product, process or resource. For example, one hears about the *metric number of function points* or the *metric lines of code (LOC) per programmer month*.
2. A scale of measurement. For example, one hears about a proposed nominal scale, i.e., classification, of software failures as a *metric*.
3. An identifiable attribute. For example, one hears about the *metric portability* of programs or the *metric coupling* in designs, even though no number or function is necessarily attached.

This confusion is unnecessary since all of the above ideas may be accommodated within the framework of scientific measurement, using the well defined terms and notions which have long been established there. For example in 1 above, *metric* is really referring to *direct or indirect measurements*.

In order to be precise, it is important that the terms be precisely defined. Following classical texts in the science of measurement (Krantz *et al.*[71], Kyburg[84], Roberts[79]), the following definitions for software measurement are used in this chapter.

Definition 4.1.1 An *attribute* is a feature or property of an entity.

Definition 4.1.2 *Measurement* is the process of empirically and objectively assigning numbers (or symbols) to the attributes of entities (including objects and events) in the real world in such a way as to describe them.

Definition 4.1.3 A *measure* is an empirical objective assignment of numbers (or symbols) to entities to characterize a specific attribute.

Definition 4.1.4 *Direct measurement* of an attribute is measurement which does not depend on the measurement of any other attribute.

Definition 4.1.5 *Indirect measurement* of an attribute is measurement which involves the measurement of one or more other attributes.

The following definitions come from (Fenton[91]).

Definition 4.1.6 A *model* is 1) an abstraction of a class of objects that retains certain attributes of those objects and ignores other attributes or 2) a mathematical formula that relates two or more measures.

For example, control flowgraphs are an abstraction of programs. Control flowgraphs retain the control attributes but throw away the data flow attributes.

Definition 4.1.7 A *product* is an artifact, i.e., a deliverable or a document, that is produced during software development.

Definition 4.1.8 A *process* is a software related activity.

Definition 4.1.9 A *resource* is an input to a process.

Definition 4.1.10 An *internal attribute* of an entity is an attribute that can be measured purely in terms of the entity itself.

Definition 4.1.11 An *external attribute* of an entity is an attribute that can only be measured with respect to how the entity relates to its environment.

4.1.2 The Need for a Foundation

This chapter is motivated by critically looking at the current state of software measurement. Software metrics and their measurements should be used to determine the state of software and to regulate and guide the development of software. However, most who design or use software metrics readily admit that our current metrics are not, in general, *reliable*, where reliability is assumed to mean accurately reflecting the real world. What is amazing is that given this acknowledged state of affairs, very few researchers are suggesting substantial changes in what we do; most suggestions for improvements represent only minor, cosmetic adjustments. But software measurement will not reach an acceptable state of reliability (in the above sense) and acceptance via minor adjustments. We have too far to go; we must make substantial changes. In this chapter fundamental questions about what we are doing, how we are doing it, and what we really need to be doing are asked and answered. In spite of our past failures we can build a foundation upon which reliable and useful metrics can be defined.

It is important to note the difference between the everyday use of *reliable* and *valid* and the use of these terms in disciplines based on measurement theory, and it is important to understand the difference between these two terms (Stanley-Hopkings[72]). A measure is *reliable* if the method used to obtain the measurement provides consistent results. * Thus, two different people using the same method to obtain the measurements will get the same value, or one person using the same method at different times will get the same value. A measure is *valid* if it accurately reflects the real world. Thus, for example, if measurements on program A and program B indicate that there is a higher degree of coupling in the code for program A than in the code for program B, then we want it to be the case that the coupling in the code for program A really is at a higher degree than the coupling in the code of program B. This is the *representation condition* of measurement as presented in the previous chapter.

* Although one does not commonly encounter discussions of the formal notion of the reliability of a software measure, this concept was introduced into the software metrics literature at least as early as 1979 (Chapin[79]).

4.1.3 The Benefits of a Foundation

Building a foundation for software measurement is important not only for software measurement but also for software engineering because with a formal foundation for software measurement we could dramatically increase our understanding of software and software engineering. Software measurement theory may be at the heart of software engineering. Such a claim may seem strange when first heard – especially given the current state of software measurement. However, when we realize that to truly measure something, one must understand what is being measured, then the claim is very reasonable. One very serious misunderstanding which we in software measurement have for sometime had is the belief that by assigning numbers to software documents, we are measuring the documents, and thus, we must be understanding the documents and their attributes. To assign numbers to a collection of entities does not produce or increase understanding. In fact, just the opposite is often the case. Assigning numbers without understanding, only leads to confusion or even worse – to an unfounded belief that one has understanding. This defining of metrics without understanding what is being defined, has led us to a situation analogous to the emperor's new clothes. We have convinced ourselves that we have developed profound and wonderful metrics; however, the simple observer sees that we have only exposed our lack of understanding.

For another argument showing that real measurement will help us better understand software engineering, consider the following. We want to measure many external factors of our software documents, e.g., reusability and maintainability, and we are instructed in our software engineering courses and by many leading software experts that by having certain internal structure such as well constructed modules and low coupling we can improve important external characteristics. Thus, we should be trying to indirectly measure or predict external characteristics based on direct measurements of internal ones. However, we do not understand how the internal characteristics influence the external ones, and thus, we are unable to determine the needed connections to develop indirect measurement and prediction tools.

4.2 Validating Software Metrics

A number of reasons have been cited for the poor industrial acceptance of software measurement. A common one is that managers simply do not know what to do with the measurements. Probably a more fundamental reason, and one which is also responsible for the conflicting claims of researchers in this area, is the lack of serious validation of proposed metrics; and thus, a lack of confidence in the measurements.

Validation when applied to software metrics has often been misused. Managers and researchers often confuse *worth of* or *ease of use* with *validity.* While not belittling these characteristics of metrics, there are differences between these characteristics and validity. Many people would consider that barometric measures of atmospheric pressure are not easy to obtain but that does not mean that they are not valid. Let us consider demonstrating the validity of metrics in the formal sense; this is a necessary requirement of all measurement activity.

By classifying software entities into *products, processes*, and *resources*, it can be shown that most software measurement activities involve the derivation of either:

1. *measures* which are defined on certain entities and characterize numerically or symbolically some specific attributes of these, or
2. *prediction systems* involving a mathematical model together with prediction procedures.

4.2.1 Measurement Activities

Components of some typical software measurement activities are shown in Table 4.1. In this table it is seen that measurements are used for *assessment* of attributes of entities which already exist (for example, for assessing the cost of a completed software project or the size of code produced) and for *prediction* of attributes of entities which may or may not already exist (for example, for the operational reliability of a software system which is currently being developed). Assessment is done via direct or indirect measurement.

4.2.2 Models and Predictions

There are essentially two different types of models which are of interest for software measurement.

Software measurement activity	Entity type	Entity example	Attribute	Measurement
Cost estimation	Process	Developing a project from specification to operation	Cost	Prediction
Productivity modeling	Resource	Personnel	Productivity	Assessment
Reliability modeling	Product	Executable code	Operational reliability	Prediction
Complexity metrics	Product	Source listing	Structuredness	Assessment

Table 4.1 *Components of typical software measurement activities*

1. There are the models of the various products, processes and resources which are necessary to define metrics unambiguously. Such models must capture the attributes being measured, e.g., a flowgraph model of source code captures certain attributes relating to control structure. Even to define a measure of an apparently simple and well understood attribute like program source-code length, we need a (formal) model of program source-code.
2. There are models which relate two or more metrics in a mathematical formula.

Let us look more closely at the latter case. Most *indirect* measures are models in this sense since an indirect measure m generally has the form:

$$m = f(x_1, \ldots, x_n)$$

where the variables $x_1, \ldots x_n$ $(n > 0)$ are measures of attributes.

The extent to which such a model is being used for *assessment* as opposed to *prediction* depends on how much is known about the variables x_i and any constant parameters of the model. In terms of measurement theory, assessment *completely* describes the attribute in question, and prediction only approximates it.

Consider a very simple example:

$$m = x/a$$

where x is a measure of source-code program length in lines of code (LOC), m is a measure of the number of hard copy pages for source-code programs, and a is a constant. If a is known to be 55 in a specific environment and if a program exists with a known x, then the indirect measure of hard copy pages computed by the formula cannot really be claimed to be solving a prediction problem. However, the object to which m relates may not yet exist nor may even the objects to which the x relates, and the constant parameters may have to be determined by empirical means. In such circumstances we are certainly using the model to solve a *prediction problem*. The more unknowns we are confronted with, the more difficult the prediction problem becomes.

Consider two well-known examples of such "models" in software measurement.

Example 4.2.1 Boehm's simple COCOMO model asserts that *effort* (measured by E person-months) required for the process of developing a software system and *size* (measured by S thousands of delivered source statements) are related by

$$E = aS^b$$

where a, b are parameters which are determined by the type of software system to be developed (Boehm[81]).

In this case to use the model for effort prediction (as it was intended) at the requirements capture stage, we need a means first of determining, i.e., predicting, the parameters a, b (Boehm gives three choices for the constants, these being dependent on the type of software system) and then of determining, i.e., predicting, the size S of the eventual system.

Example 4.2.2 The Jelinski-Moranda model for software reliability prediction assumes an exponential probability distribution for the time of the i^{th} failure (Jelinski-Moranda[72]). The mean of this distribution (and hence the Mean Time to i^{th} failure) is given by

$$MTTF_i = \frac{\alpha}{N - i + 1}$$

where N is the number of faults assumed to be initially contained in the program and ϕ $(= 1/\alpha)$ represents the *size of a fault*, i.e., the rate at which it causes a failure. Faults are assumed to be removed on observation of a failure, and each time a fault is removed the rate of occurrence of failures is reduced by ϕ. The unknown parameters of the model, N and α, have to be estimated by some means, for example by using Maximum Likelihood Estimation after observing a number of failure times.

In both the previous examples, the model alone is not enough to perform the required prediction. Additionally we need some means of determining the model parameters, and a procedure to obtain the results. Thus, in the case of the COCOMO cost prediction model, the parameters are determined by a combination of calibration according to past history, expert judgement, and subjective judgement; the theory provides various means of interpreting the results based on a plug-in rule for the parameters. In the case of reliability growth prediction models, it is not enough just to have a probabilistic model such as Jelinski-Moranda, but one must also have a *statistical inference procedure* for determining the model parameters and a *prediction procedure* for combining the model and the parameter estimates to make statements about future reliability (Jelinski-Moranda[72]). With these observations, Littlewood's definition (Littlewood[89]) of *prediction systems* is used to encompass the notion of the underlying *model* and the prediction procedures.

Definition 4.2.3 A *prediction system* consists of a mathematical model together with a set of prediction procedures for determining unknown variables and parameters.

Thus, it is incorrect to talk about **the** COCOMO method (or "metric") for cost estimation or **the** Jelinski-Moranda model for reliability prediction since any results obtained on the same data will vary according to the particular procedures used. We should instead talk about COCOMO cost prediction systems and J-M reliability prediction systems.

A model defines an association between attributes, and a prediction system illuminates the nature of that association. However, in some cases it is not necessary to determine the full functional form of the relationship; a procedure for confirming the existence of the association and certain properties of the association may be sufficient.

Example 4.2.4 Kitchenham, Pickard and Linkman confirmed an association between large values of certain program structures and size measurements on one hand and large values of measures of fault-proneness, change-proneness, and subjective complexity on the other hand (Kitchenham *et al.*[90]). Thus, they were predicting fault-proneness, change-proneness and subjective complexity based on their structure and size measurements. As a result of their study, a special type of prediction system was constructed. This had two components:

1. a procedure for defining what was meant by a *large* value for each measure and
2. a statistical technique for confirming that programs with large values of size and structure metrics were more likely to have a large number of faults, and/or changes, and/or to be regarded as complex than programs that did not have large values.

Kitchenham *et al.* concluded that their procedure of identifying programs with large values of size and/or structure could be used to assist project managers to reduce potential project risks by identifying a group of programs or modules which were likely to benefit from additional software development, for example redesign or additional testing. However, they also pointed out that because the associations were quite weak:

1. some programs with large size and structure values would not exhibit any problems, so that additional effort spent on their development would be wasted and
2. many programs which later exhibited problems did not have large size or structure values.

There is often confusion about the applicability of proposed models and metrics. Since it is often believed that the ultimate goal of software measurement is *prediction*, proponents of a particular metric have often claimed that their measure is part of some prediction system when in fact it might simply (and usefully) be a measure of an interesting attribute as opposed to a predictor.

Example 4.2.5 Albrecht's Function Points (FPs) (Albrecht[79]). It seems that these were intended to measure the attribute of *functionality* in specification documents as perceived by a user. This is a product measure, and to validate it as such would require confirmation that it captures (at least) the intuitive order of specifications with regard to their amount of functionality. This is, if specification S_1 has more functionality than specification S_2, then the function point measure for S_1 must be greater than that for S_2. (Unfortunately, they are defined in such a way that attributes such as the systems analysts' view of complexity rather than the user view of functionality is also taken into consideration by the weighting factors.) A validated measure of such an important product attribute would be very welcome. However, it would appear that nobody has ever attempted to validate the FP measure with respect to this intuitive order. In fact, FPs are not used as measures. They are used as part of a cost prediction system, and when people refer to the validation of FPs, they implicitly refer to validation of this predictive theory.

4.2.3 Formal Validation

There *many* software metrics in the literature. Not only do these aim to measure a wide range of attributes, but also there are many irreconcilable metrics all claiming to measure or predict the same attribute such as cost, size, or complexity. In this section the necessary validation procedures which could at least partially resolve many problems are described.

In the software measurement literature *software metrics* may refer to *measures* or *prediction systems*. Informally, a *measure* is valid if it accurately characterizes the attribute in question; a *prediction system* is valid if it makes accurate predictions. The approaches to validation in each case, starting with prediction systems, will be described. In the case of validating measures there is often an implicit assumption that a measure must always be *part of* a prediction system to be useful; the pitfalls of this assumption are examined. Proper validation approaches and traditional validation approaches are compared, and ways to avoid typical mistakes in validation experiments are given.

Validation of Prediction Systems

Definition 4.2.6 *Validation of a prediction system* in a given environment is the process of establishing the accuracy of the prediction system by empirical means, i.e., by comparing the model's performance with known data points in the given environment.

Thus, validation of prediction systems involves experimentation and hypothesis testing. It is not analogous to mathematical proof, but rather it involves confirming, or even failing to disprove, a hypothesis.

This type of validation is reasonably well accepted by the software engineering community. For example, it is what is normally meant when one hears about an attempt to validate the COCOMO model or a particular software reliability growth model. As a specific example Abdel-Ghaly, Chan, and Littlewood describe a procedure for evaluating software reliability growth models in such away that the *most valid* in a given environment can be determined (Abdel-Ghaly *et al.*[86]). These ideas are taken further in (Brocklehurst *et al.*[90]) where by using techniques of recalibration a new prediction system for software reliability prediction (using a multi-model approach) is shown to be *valid*.

Just how accurate a prediction system has to be before we accept it as validated will in many cases depend on the person using it. However, there seems to be a distinction between prediction systems which are *deterministic* with regard to the underlying model (meaning that we always get the same output for a given input) and those which are *stochastic* (meaning that the output for a given input will vary probabilistically).

Example 4.2.7 Suppose that we count LOC by number of carriage returns. If we know that a printer prints 55 lines to a page, then the prediction system for computing number of pages m based on the model $m = \text{LOC}/55$ is deterministic. There should be no deviation at all in the result. In fact, we should probably not call this system a *predication* system.

On the other hand, if we used the same model where LOC is defined by

$$\text{LOC} = \frac{\text{Total number of characters}}{35}$$

(where 35 is the *average* number of characters per line), then the prediction system for number of pages is stochastic. We would expect any prediction to be accurate only within certain bounds, to allow for the fact that the number of characters on a line is a random variable.

Some *stochastic* prediction systems are more *stochastic* than others, i.e., the error bounds are wider. The last example is much *less* stochastic than a prediction system for *number of failures* which is based on the model

$$\text{Number of failures} = \frac{\text{LOC}}{100}.$$

Prediction systems for software cost/effort estimation and reliability are clearly *very* stochastic, i.e., the margins of error are large. Boehm has claimed that under certain circumstances the COCOMO effort prediction system will be accurate to within $\pm 20\%$. This specifies the acceptance range for validation. When no such range has been specified, it is up to the user of a prediction system to specify in advance what range is acceptable.

A common mistake in software measurement is to formulate stochastic models as if they were deterministic. COCOMO is again a good example. Boehm suggests three pairs of values for the a and b parameters used in his equation. When empirical studies identify different values, we should not reject the COCOMO cost prediction system but rather suggest that the new values be used to *calibrate* the model. This may be regarded as a minor point, but lack of appreciation of the stochastic nature of most of our prediction systems means that people forget that estimates of the parameter values are themselves subject to error. This is a particular problem when cost estimators want to use the value b to identify whether or not there are economies or diseconomies of scale in a software environment.

For example, Banker and Kemerer investigated possible reasons for economies and diseconomies of scale by drawing inferences from the b values found in 9 data sets although they themselves provide the analysis which shows that only 2 data sets had values of b significantly different from 1 (Bankar-Kemerer[89]). Their final analysis may be reasonable, but their original assertion that there appeared to be a relationship such that values of $b < 1$ were associated with data sets which involved small projects and values $b > 1$ were associated with data sets which involved large projects seems to be unjustified.

Validation of Measures

Metrics used for assessment are simply measures in the sense of measurement theory. Hence we turn to measurement theory for the appropriate notion of validation.

Definition 4.2.8 *Validation of a software measure* is the process of ensuring that the measure is a proper numerical characterization of the attribute in question; this means showing that the representation condition is satisfied.

Example 4.2.9 A valid measure of the attribute of **module coupling** of designs must not contradict any intuitive notions about module coupling. Specifically, it requires a formal model for designs (to enable objectivity and repeatability) and a numerical mapping which preserves any relations over the set of designs which are intuitively imposed by the attribute of module coupling. Thus, a proposed measure of module coupling should indeed measure precisely that; in particular, if it is generally agreed that design D_1 has a greater level of module coupling than design D_2, then any measure m of module coupling must satisfy $m(D_1) > m(D_2)$. A specific measure of module coupling which is valid in this sense is proposed in (Fenton-Melton[90]).

This type of validation is based on the representational theory of measurement. Practitioners may prefer to regard this as ensuring the well definedness and consistency of the measure. Examples of how this approach may be used to define and validate measures of specific internal product attributes like *structuredness*, *coupling, modularity*, and *reuse* are described in (Fenton[91]).

4.2.4 How "Validation" Has Been Performed

There is an implicit assumption in much of the software engineering community that validation (in the measurement theory sense) of a measure is not sufficient. Specifically, it is expected that *validation* must also entail the demonstration that the measure is itself part of a valid prediction system.

Some people argue that LOC is not a *valid* software measure because it is not a good predictor of reliability. However, whether LOC is or is not a good predictor of reliability is not the deciding factor as to whether LOC is a valid software measure.

It is often assumed that a measure should be an accurate predictor of some software attribute of *general* interest. These are assumed to be certain process attributes like cost, certain external product attributes like reliability, and certain resource attributes like productivity. A typical harmful ramification of this assumption is to reject as *invalid* perfectly good measures of, for example, internal product attributes (like size, structuredness, and modularity) on the grounds that they do not measure something which is of sufficient interest.

Formalizing this commonly accepted assumption would lead to a new definition of validation for a measure. Let us say that a measure is valid *in the extended sense* if it is

1. *internally* valid (meaning valid in the measurement theory sense) and
2. a component of a valid prediction system.

Let us suppose that we wish to show that a measure is valid in the extended sense. Then what we really need is a hypothesis proposing a specific relationship between our measure and some *useful* attribute. We would then need to conduct an experiment to gather the relevant data to test the hypothesis.

Unfortunately, what has often happened is a measure is shown to be valid or invalid in the extended sense by correlation against any *interesting* measurements which happen to be available as data. For example, a measure of modularity might be claimed to be valid or invalid on the basis of a comparison with known *development costs* if the latter data is available. This would be done even though no claims were ever made about a relationship between modularity and development costs.

It is perhaps conceivable that a measure could be shown to be valid in the extended sense even though no hypothesis initially existed. For this to happen, the data which happen to be available would have to be shown to be consistently related via a formula determined, for example, by regression analysis. However, given our poor understanding of software, this type of validation appears fraught with difficulty. *And yet many software engineers have assumed this as the major approach to validation.*

It may be of some comfort to the many researchers who have taken this approach to note that they have not been alone in making these mistakes. Indeed, speaking quite generally about measurement validation, Krantz in his excellent formal treatment of measurement (Krantz *et al.*[71]) asserts:

> A recurrent temptation when we need to measure an attribute of interest is to try to avoid the difficult theoretical and empirical issues posed by fundamental measurement by substituting some easily measured physical quantity that is believed to be strongly correlated with the attribute in question: hours of deprivation in lieu of hunger; skin resistance in lieu of anxiety; milliamperes of current in lieu of aversiveness, etc. Doubtless this is a sensible thing to do when no deep analysis is available, and in all likelihood some such indirect measures will one day serve very effectively when the basic attributes are well understood, but to treat them now as objective definitions of unanalyzed concepts is a form of misplaced operationalism.

However, measurement and prediction are not completely separate issues. On the contrary, as Kyburg observes (Kyburg[84]):

> If you have no viable theory into which X enters, you have very little motivation to generate a measure of X.

The initial obligation in proposing metrics is to show that they are valid in the narrow sense. *Good predictive theories only follow once we have rigorous measures of specific well understood attributes.*

Another common approach taken to validating new metrics is to show that they correlate with some well known existing metrics. An example of an extensive validation of this sort is given in Li and Cheung (Li-Cheung[87]). The common notion that metrics such as Halstead's, McCabe's, and LOC are in some sense validated (in the extended sense) is a historical aberration resulting from a misunderstanding of the obligations of true measurement. While these might be valid metrics of very specific attributes (like *number of decisions* in the case of McCabe and source code program *length* in the case of LOC), they are not valid metrics of things like cognitive complexity, correctness, and maintainability. Although there is empirical evidence to suggest that these metrics are *associated* with development and maintenance effort and errors, such correlations in no way imply that these are good predictors of these attributes. The many claims made to the effect that metrics like Halstead's and McCabe's have been validated as part of prediction systems are equaled by studies which show that they correlate no better with the process data than a simple measure of size like LOC (Hamer-Frewin[82]).

A more compelling reason why we have to be very wary of the *correlate against existing metrics* approach is that unstructured correlation studies run the risk of identifying spurious associations. For example, using the 0.05 significance level, we can expect a significant but spurious correlation 1 in 20 times *by chance*. This means that if you have 5 *independent* variables and look at the 10 possible pairwise correlations, there is a 0.5 (1 in 2) chance of getting a spurious correlation.

Examples of Validations

Example 4.2.10 Between 1986 and 1988, one of the authors was involved in a UK government (Alvey) funded collaborative project

(Fenton[90]). This project suffered from some of the above misconceptions surrounding validation.

It was already known to the project team that their results on the theory of structure had provided a useful tool for analysis during maintenance. They had metrics of various internal structural attributes. Without even considering whether these metrics were valid and without any hypotheses or proper experimental framework, they attempted to correlate values of the metrics for specific modules or programs against historically collected data. The data were process-type metrics, e.g., cost, changes made, etc., which *happened to be available* for the modules and programs. Not surprisingly, the results were the same as those of many previous studies, i.e., all the metrics correlated reasonably well with each other, and none correlated with the data significantly better than lines of code (LOC).

In this case there should have been some specific hypotheses proposed as the basis for a prediction system. Based on their knowledge of the applications of the metrics, the project members should have started by attempting to show that a number of metrics of different structural properties were specifically related to the *maintainability* of the resulting code. This would have required collecting data which was specifically relevant for maintainability, such as time to find and repair bugs. Due in part to a fear that placing emphasis on maintenance could be seen as a negative selling point, this kind of hypothesis was never properly articulated.

There are some additional observations which should be noted from the experiences on this project and which appear to be generally relevant.

1. Historical data is notoriously unreliable, partly because there are no agreed upon standards for defining the process-type metrics.
2. If one believes that a measure correlates with a specific attribute, like for example *reliability*, then it does not help if the only data are historical metrics of say *cost* and *changes to modules*. What one would need is some data relating to *failures*.
3. Ideally, what is needed is first to formulate the hypotheses and then design experiments *before* data collection. If there is no alternative than the use of historical data, then at least the hypothesis should be formulated before the data is analyzed.

Example 4.2.11 A very simple example of a software measure is the much quoted lines of code metric, LOC. Is this a valid measure?

In the measurement theory sense it is indeed valid; it is defined on a specific software product – code – and suitable formal models exist to allow an unambiguous formal definition of LOC. There is a very well-known attribute of the code which is captured by LOC, namely, *length*. This is a useful attribute to know; for example, it gives a clear indication of how much paper will be required for a print out and how many disks will be required for its storage. (Strictly speaking of course models and prediction systems are required to make even these simple assumptions formal.) Length is an attribute which imposes intuitively understood and accepted empirical relations on all instances of code. People can generally reach agreement on which of two pieces of code is longer, and the representation condition assures us that LOC is a valid measure of length if it preserves this relation. And it certainly will preserve the relation (given a suitably rigorous definition). Thus LOC is a valid measure of length.

However, LOC has not been shown convincingly to be a valid measure of *complexity*, nor has it been shown to be part of an accurate prediction system for complexity. Complexity is an ill-defined attribute. In the software engineering literature it is assumed to be an attribute which in itself impacts a wide range of other attributes like reliability, maintainability, and cost.

Example 4.2.12 In a recent paper (Fenton-Melton[90]), there is an example of how a previous "validation" study would have benefited from a consideration of the two types of validation discussed here. The study by Troy and Zweben attempted to show that *coupling* was a valid measure for designs and programs (Troy-Zweben[81]). Given the data which they used to perform validation, their implicit hypothesis can be reconstructed.

> Designs (programs) with high coupling contain more errors than designs (programs) with low coupling, i.e., "quality" measured by number of discovered program errors is inversely proportional to the level of coupling.

The problem is that Troy and Zweben have not proposed a specific measure of coupling, and without a proposed measure of coupling the hypothesis is untestable. What Troy and Zweben did was attempt a linear correlation of all the various "counts" which they felt might contribute towards coupling (like number of modules, total number of module connections, etc.) separately against number of errors recorded for various programs. This did not test the hypothesis above.

With a measure of coupling as described in (Fenton-Melton[90]), a more interesting hypothesis (which gets rid of many of the experimental design flaws) could be properly tested:

> For programs of similar length, those with higher coupling contain more errors.

In fact, a number of other similar hypotheses could be tested. For example, *contain more errors* could be replaced by *are more difficult to maintain*. Given a good measure of functionality, one could refine the hypothesis by adding *and of similar functionality* after *length*. What is interesting about this approach is that it may lead to metrics of coupling and the like being used constructively in design. We should dispel the idea of simple linear correlations between coupling and errors. It seems far more plausible that for a given size program, there is a stochastically *optimal* level of coupling with regard to likely errors (and this optimal level will certainly not be zero as is implied by simple linear correlations). It is these kinds of numerical standards which may be achievable.

Choosing Appropriate Prediction Systems

In order to help formulate hypotheses necessary for validating the predictive capabilities of metrics, it is necessary to be clear about the different types of prediction systems which are appropriate. Such systems can be divided into the following classes:

1. Using internal attribute measures of early lifecycle products to predict metrics of internal attributes of later lifecycle products, e.g., metrics of size, modularity, and reuse of a specification to predict size and structuredness of the final code.
2. Using early lifecycle process attribute measures and resource attribute measures to predict measures of attributes of later lifecycle processes and resources, e.g., number of errors found during formal design review to predict cost of implementation.
3. Using internal product attribute measures to predict process attributes, e.g., metrics of structuredness to predict time to perform some maintenance task or number of errors found during debugging.
4. Using process metrics to predict later process metrics, e.g., metrics of failures during one operational period to predict likely

failure occurrences in a subsequent operational period. In examples like these where an external product attribute (reliability) is effectively defined in terms of process attributes (operational failures), we may also think of prediction systems as process metrics being used to predict later external product metrics.

It may be difficult to design prediction systems relating internal structural attributes to external and process attributes. However, it is generally assumed that certain internal attributes which are the result of modern software engineering techniques, e.g., modularization, low coupling, control and data structuredness, information hiding, and reuse, will generally lead to products which exhibit a high degree of the desirable external attributes like reliability and maintainability. Thus, programs and modules which exhibit poor values for the desirable internal attributes, e.g., large unstructured modules, are likely (but not certain) to have more faults and take longer to produce and maintain. A sensible manager will develop some procedures for handling such components before they become problems, e.g., break them up into smaller items or give them extra review and testing effort. This is using measurement for *risk reduction*, which in this case amounts to redistributing development effort to the areas which are most likely to benefit and fits in with a decision theoretic framework.

4.3 Summary

Software measures and prediction systems will not be widely used or respected without a proper demonstration of their validity. However, commonly accepted ideas and approaches to validating software measures bear little relation to the requirements of validation for measurement in other disciplines. Specifically there are formal requirements for validation which we must first address before we can hope to tackle the informal notions like *usefulness* and *practicality.* These requirements are:

- In the case of measures — justify that the measure characterizes the stated attribute in the sense of measurement theory.
- In the case of prediction systems — decide how *stochastic* the prediction system is, i.e., what are the acceptable error ranges, and conduct experiments to compare performance of the prediction system with known data points.

Software metrics which characterize specific attributes do not have to be shown to be part of a valid prediction system in order to be valid measures. A claim that a measure is valid because it is a good predictor of some interesting attribute can only be justified by formulating a hypothesis about the relationship. This amounts to validating a proposed prediction system.