Chapter 2

Software Metrics In Project Management

2.1 Historical Overview

In this chapter, we first trace the history of software metrics and related research, and then analyze their impact and acceptance by the software industry. We then present the test and evaluation metrics set, that provides the core set of metrics which can be used for monitoring and managing large software projects. Software metrics are used to measure and reduce software project costs, improve software productivity, and most importantly, improve the quality of products. The attributes that are generally monitored include the following:

- *Project*: Project costs, manpower, resource utilization, personnel productivity, estimation of completion time, perceived risk due to development delays, etc.
- *Process*: Process size, performance complexity, reusability performance characteristics, etc.
- *Product*: Quality, estimated residual errors still undetected, reliability, maintainability, enhancibility, etc.

The applicability of metrics span far beyond the restrictive boundaries portrayed above. One can have enterprise-wide or industry-wide metrics, as well as metrics that indicate the user's satisfaction, with regard to the software product [ROY87]. The science and engineering of metrics thus include measurements on processes, projects, products; as well as their analysis and evaluation. Metrics data capture and encapsulate the previous experiences, help develop models and theories of the observed phenomena, and thereby support prediction of project, product and resource requirements.

The concept of instituting discipline into the software development and the emphasis on the use of metrics are not new. Considerable research has been conducted in software engineering in the past two decades, especially on reliability models, cost estimation, and application of software metrics.

Hewlett-Packard (HP) in the late sixties also emphasized process improvement. HP emphasized process improvement via process instrumentation and metrics as opposed to purely software testing [GRA87]. The historical sequence of metrics development is as follows:

1. Project management based metrics started in the early 1970s.
2. Metrics based on system evolution started around the mid 1970s.
3. Software science (product) based metrics started around the mid 1970s.
4. Reliability and quality based metrics started around the late 1970s and early 1980s.
5. Complexity (product) based metrics started around the late 1970s and early 1980s.
6. Cohesion-coupling metrics started in the late 1970s.
7. Requirements and design stability metrics started in the middle and late 1980s.
8. Integrated process based comprehensive metrics started in the early 1990s.

The above classifications reflect:

1. The immediate concerns of the time. For example, project metrics were important to aid cost estimation and control.
2. The evolution of software technology, and
3. The increased awareness and insight into the problems of software quality, reliability, complexity and maintenance.

2.1.1 Cost and Effort Estimation Models

The earliest application of metrics was concerned with project costs and efforts. Pioneering work at IBM [WAL77], US Army [PUT92], and TRW [BOE78] emphasize monitoring cost-effort data obtained in similar software projects and using them to predict cost and effort estimates on new projects. Many of the studies modeled the cost-effort characteristics as a statistical distribution [VOS87], and the predictions were derived from measured information. A similar but more parameterized and practical approach was later used in the Function Point Analysis [ALB83]. The main drawbacks of these approaches are that they do not take into account the product life cycle costs and efforts. Because of their primitive nature, they are not effective in newer, more advanced development environments like the current PC-based client server collaborative configurations. They also depended upon a questionable unvalidated assumptions about statistical distributions.

2.1.2 Metrics Based on System Evolution

Belady and Lehman [BEL79a] studied the evolution of the large software system IBM OS/360. This large operating system vertically integrated several architectural models of IBM System 360. It involved thousands of software professionals working at different locations. Hardware and software were co-designed and co-evolved, even though it started out with the "hardware-first" rule, ie. hardware architecture is specified and frozen first, and its software is designed and developed next. Software development for OS/360 faced several problems such as:

- changing requirements during product development

- numerous modifications to design
- continuous improvements and hence changes to the programming language
- incomplete prototyping and testing because of product release deadlines, that is, no alpha or beta testing.

Based on the software metrics gathered over six years, Belady and Lehman postulated three major "laws" of software evolution [BEL79a]:

1. *The Law of Continuous Change.* This postulated that software systems are continuously modified because of their ease of change.
2. *The Law of Increasing Entropy.* This implies that "change over change" induces unstructuredness, which in turn leads to irreparable or unmaintainable products.
3. *The Law of Statistical Stability.* This implies that under proper control, the software product may ultimately sustain a period of stability, i.e. no more changes. Belady and Lehman called this the state of *statistical equilibrium.*

The first two laws, while empirical, appear to hold both intuitively and in actuality. The third law appears to be questionable since there is no substantiating data.

Belady and Lehman recognized the effect of requirements and design perturbations in their study of the OS/360 system. They noted that changes impacted several modules up and down the design stream. Metrics data identified the effects of unstructuredness in the code due to repeated changes. It showed the importance of partitioning and modularization to contain the adverse effects of unstructuredness. It helped to propel structured design and structured programming into general use.

Yau and Collefello [YAU85] studied the code changes made during debugging and maintenance and the ripple effects they caused in the development cycle. Using

dependency arguments, they identified code sectors that need to be evaluated or modified using stability criteria (program dissection or program slicing). Kafra and Reddy [KAF87] used similar arguments to develop techniques to assess the code stability in order to contain the adverse effects of requirement and design changes.

2.1.3 Software Science (Product) Metrics

Software science metrics evaluate the gross characteristics of the software such as size (number of executable lines of code), program volume, etc. It is due to Halstead who theorized that computer programs can be represented by a sequence of tokens consisting of operands and operators [HAL75,HAL79]. He considered it a science, because it depended on measurements based on visible product characteristics. He hypothesized that when comparing the same function programmed in two different computer languages, the operator-operand ratios or densities in the program will be approximately the same, i.e. the information content, in the information theory sense, will be the same. The basic metrics of software sciences are:

n_1: Number of unique operators in the program

n_2: Number of unique operands in the program

N_1: Total number of operators in the program

N_2: Total number of operands in the program

The basic measures of program complexity are:

Program Length: $N = N_1 + N_2$

Program Effort: $E = V / L$

where

v (Program Volume) = $N \log_2 n$

n (Program Vocabulary) = $n_1 + n_2$

L (Program Level) = V^* / V

where V^* = volume of the most compact design implementation

The estimated length of the program is:

$n' = n_1 \log_2 n_1 + n_2 \log n_2$

Software science metrics are product based metrics which are easy to calculate. They are applicable to all languages but could be language sensitive. It is based partly on the information theoretic viewpoint and not practiced extensively.

2.2 Test and Evaluation Metric Set

In this thesis, we propose a set of test and evaluation metrics [PAU93,HEN92,PAU94b] that deals with various management, requirement, and quality attributes of computer system software. These metrics serve as measures and indicators that critical technical characteristics and operational issues of both software and the integrated system have been achieved. Collection of additional metrics of specific interest will also be useful in monitoring or supporting a particular agency's needs, and will also aid in more accurate advice generation.

2.2.1 Objectives

The objective of the test and evaluation metrics is to integrate test and evaluation into the software engineering development process to ensure that:

- the software process is kept under control, so that software progress can be demonstrated at specified intervals; and

- immature software is prevented from entering the user system tests or being deployed.

In particular, measurement enables engineers to

- quantify product reliability and performance,
- to isolate development process and product attributes that impact reliability and performance, and
- to demonstrate how process and product changes impact these attributes.

Measurement also lets development teams:

- set achievable goals
- demonstrate their potential to meet these goals
- track their progress
- adjust processes to correct out-of-bounds conditions, and
- demonstrate the impacts of these adjustments on meeting goals.

The metric values identify high risk values in software development and maintenance, notifying management to allocate attention and resources required for control of these items. Data collected for the test and evaluation metrics should be presented at major system milestone reviews. The selected metrics set allows software engineers and managers to identify the level of risk associated with each significant issue in the software life cycle. When properly used and adequately analyzed, it can provide a continuous measurement process for:

- the management process for software development;
- requirements definition and formal specification; and
- the quality of software products.

2.2.2 Selection Criteria

The minimum level of information which is required for a software risk management program is:

1. Understanding of the risk factors and the system user's requirement for risk control;
2. Measures (software metrics and measurable management issues) to identify and measure risk factors and to specify users' risk requirements;
3. Test resources to derive risk measurements;
4. Tools to extrapolate raw measurement data into useful management information.

The test and evaluation metrics set was developed to provide information which would allow managers to manage the most common risks which are encountered in software development programs. Review of the literature identifies the following common risk issues in software development [PAU94b,RAM85,BOE78]:

1. Inadequate statement of user requirements
2. Operational failures of system software
3. Change of software user requirements
4. Inadequate software test and evaluation
5. Excessive changes to software design
6. Cost overrun
7. Schedule slip
8. Incomplete software problem resolution during development
9. Overrun of physical computer resources
10. Inadequate capability of the software developer.

The test and evaluation metric set is selected to address the entire software development and test and evaluation process by providing both process and product measures. Selection criteria for the metric set include the following [BAS84]:

- Is unambiguous;
- Has methods for data collection and evaluation described;
- Has non-labor-intensive data collection and evaluation methods;
- Is capable of consistent interpretation and is in formats that are objective, timely, and finite
- Has intrinsic worth and demonstrate added value to the software development process
- Supports risk management for one of the common risk areas listed above.

2.2.3 Customer Satisfaction Measures

The selection of the core set of test and evaluation metrics is based aimed at measures that achieve customer satisfaction. The product of the software development process must be of high quality and must satisfy the customer. "Quality" in this sense can be further divided into:

- *in-process quality*, which refers to quality control during the software development process before product shipment;
- *release quality*, which refers to defects discovered between functional tests and shipment;
- *service quality*, which refers to the servicing of defects for the first six months after product release. This often-overlooked type of quality of one component of customer satisfaction.

Despite this popular definition of quality, it is well known that the absence of defects does not necessarily build business. Customers are not only interested in products that have no defects; they are interested in products that fulfill their requirements and improve their productivity. The developer must therefore not only be aware of customers' requirements and expectations but also show constant improvement, especially to take advantage of new technology to make the job easier [HAM90]. The ultimate aim of the development system is therefore to provide products that satisfy the customer.

Furthermore, it is not simply sufficient to have happy and loyal customers. The developer must constantly keep up-to-date with changing customer requirements as well as the competition. The customer expects only what the producer has led him to expect. Modern-day customers have a tendency to compare products and sources. Even satisfied customers will switch if they can find alternative products that are better suited to their particular requirements.

Users want applications that:

- can be implemented quickly
- function at a level that they need and understand
- require little maintenance effort
- require little translation between the languages of programmers and users
- are responsive to changes in their needs.

2.2.3.1 Meeting Customers' Requirements.

If the aim of the development system is to products quality products that meet customers' requirements, then it is not only sufficient to follow quality procedures and develop quality products; the system must support managers and developers with quality

procedures to develop quality *products that meet customers' expectations.* This objective must be clear to both management and development personnel, not only for the current product being developed but also for the development of future products. The components of the system must cooperate and must be managed. The objective of the system is to produce something of value, that is, it must produce results within current cost constraints. Objectives are therefore defined in terms of results rather than in terms of a specific activity or method. Collecting software metrics of a project and designing a decision analyzes support system to access and analyze the metrics in the database provides a quantifiable means of measuring results which is superior to conventional qualitative ad-hoc measures [PAU93].

Requirements must be:

- Viable with today's technology
- Credible
- Based on an understanding of the needs of the application
- Competitive with those of products they are up against
- Have synergy between the provided functions.

Furthermore, individuals and groups developing requirements must:

- Understand the application needs
- Have knowledge of the solution technologies
- Be respected for knowledge of requirements
- Communicate effectively
- Successfully lead the implementation effort.

The development process is not static but can instead benefit from customer and management feedback, as well as from the lessons of other related projects [PAU92]. The system should therefore also include a feedback loop for continual improvement of product or service, and continual learning. Again, identification and collection of a core set of useful metrics and their analysis constitute an invaluable form of feedback. Management and developers can make use of such information to observe the effect of redesign on cost, sales, and evaluation by customers . Based on such feedback, the system can also predict what components of the system will be affected and by how much as a result of proposed changes in one or more components.

2.2.4 Classification

Based on the above discussion, therefore, the set of test and evaluation metrics thus identified can be divided into 3 categories, namely *management metrics, requirements metrics* and *quality metrics.* This set of metrics comprise a minimum set for information gathering over the entire system software life cycle. The justification for the classification is based on the differing views and needs of four important components of the system procurement: management, users, developers and maintainers.

2.2.4.1 Management Metrics

These metrics are used to support management in the control the software development, by the selection of high quality development personnel, setting realistic schedules and providing adequate computer and other resources within budgeted costs.

2.2.4.2 Requirements Metrics

As described in the previous section, one of the ultimate measures of the success of a software product is customer satisfaction: the product must meet customer

requirements. This set of requirement metrics is therefore designed not only to ensure that the products thus developed meet customers' initial requirements, but how well the software development process can respond to continual change in customer requirements during the software development process. Excessive changes in requirements may indicate the inability on the part of management to use their foresight to predict customers' new requirements. Management works directly with customers to determine their needs, and must look ahead and use their foresight to design products that meet customer requirements. Requirements definition and refinement occurs over the entire development cycle, and must be integrated into the development process to ensure that customers' needs are met.

2.2.4.3 Quality Metrics

These metrics are required to ensure the development of a quality product with a minimum of bugs that meet customers' requirements. They are mainly intended to meet the developer's needs. Software developers typically have little knowledge of the end user and are out of contact with real application users. A common reason for failures of large software projects is that developers misinterpret management communication of users' requirements or other subsystem interfaces and develop products that do not meet end users needs. It is also a habit of developers to treat requirements as being explicit and complete rather than as examples of a more general need. If an application solution is designed for specific requirements, the resultant product will be very specific and inflexible. To minimize this possibility, quantitative measures must be in place to ensure that end user needs are constantly taken into account during the software development process. In addition, since large software products have a long life-span due to their high

costs, measures must also be in place to ensure that the product is easy to maintain and enhance even after it is released to customers. Complexity metrics that measure the degree of complexity of the product as well as the interactions between modules of a product are useful in this respect in ensuring that the software is not unnecessarily difficult to maintain.

As a result of the above discussion, we conclude that some of the predictors of customer satisfaction which should be incorporated as part of our test and evaluation metric set are as follows:

- Schedule requirements
- Requirements complexity
- Existence of implementation technology
- Marketplace and knowledge of intended customer
- Projected product size and defect density
- Management maturity
- Level of innovation required
- Culture of user/customer
- User/customer vision

The metrics database is constantly evolving and is expected to change as organization needs change. In addition to the three classes of software metrics identified above, the collection of additional metrics of specific interest will also be useful in monitoring or supporting a particular agency's needs, and will also aid in more accurate advice generation.

2.2.5 Formal Notation for Metrics Domains

For the purpose of formal specification for the remainder of this report, we use the abbreviations illustrated in Figure 2.1 to denote the metrics that fall under the various categories.

Each of the test and evaluation metrics discussed consist of several measures or data elements. Each data element characterizes some attribute of the overall metric. More formally, a metric can be defined as a *relation* R with several *attributes* A_i, i=1,...,n. Each attribute comprises a *domain* D_i, i=1,..,n, as follows:

$$R \subset D_1 \times D_2 \times \ldots \times D_n$$

Abbreviation	Metric
Management Metrics	
Co	Cost
Sc	Schedule
Cr	Computer Resource Utilization
Se	Software Engineering Environment
Requirements Metrics	
Rt	Requirements Traceability
Rs	Requirements Stability
Quality Metrics	
Ds	Design Stability
Cm	Complexity
Bt	Breadth of Testing
Dt	Depth of Testing
Fp	Fault Profiles
Re	Reliability

Figure 2.1: Metrics Classification and Abbreviations

Metrics databases are not static but are evolutionary in nature. As the organization evolves, management needs change, and new metrics may be collected or new attributes may be added to existing metrics. In general, adding an attribute to an existing metric

results in adding a new domain to our relation R. Denote the existing domains $D_1 \times D_2 \times \ldots \times D_n$ by Ç. Then, suppose we define an operator α such that a new domain is added:

$$\alpha_k : Ç \rightarrow Ç \times D_{n+k}$$

We can then rename $R_k = Ç \times D_{n+k} \rightarrow Ç$, which the current state of our metrics database with the new attribute added. We will use this notation throughout this chapter to formally represent domains for various metrics.

2.3 Related Work on Metrics Databases

In this section we survey more recent advances in software metrics, theories, measurements, tools and methodology. Jablonowski [JAB94] explored the use of fuzzy set theory to express the element of vagueness embodied in the intuitive definition of risk. Grey [GRE92] discussed the benefits of quantitative modeling techniques, including probabilistic models for software cost and schedule risk analysis.

Software metrics have also been used as an aid to management decisions in large corporations. Hewlett-Packard [GRA94] uses four classes of software metrics to aid in management decision making:

- project estimation and progress monitoring
- evaluation of work products
- process improvement through failure analysis
- experimental validation of best practices.

Grady also recommended that project managers collect the following data:

- engineering effort by activity
- size data

- defects counted and classified in multiple ways
- relevant product metrics
- complexity, and
- testing code coverage.

These data are collected in support of the Functionality, Usability, Reliability, Performance and Supportability (FURPS) criteria: functionality, usability, reliability, performance, and supportability. Managers should then understand how each of these relates to their successes, and perform timely analyses to optimize future projects.

The crucial dependency of NASA's spacecraft and ground systems on software to meet mission objectives resulted in the initiation of a software metrics program in 1990 by NASA's Mission Operations Directorate (MOD) [STA94]. The metrics toolkit helped MOD to determine whether a project was on schedule and within budget, when subsystems were ready to be integrated, and whether a particular test schedule made sense. The toolkit helped ensure consistent data collection across projects and increased the number and types of analysis operations available to project personnel. The toolkit also helped engineers and managers make decisions about project and mission readiness by removing the inherent optimism of engineering judgment.

Defect data from inspections and test were key elements of process improvement initiatives that Bull HN Information System's Enterprise Servers Operation in Phoenix, Arizona, started in 1989 [WEL93]. By examining data from several projects, project members see how data is used to better understand a development project's dynamics. Feedback from the project metrics is used to improve processes. Feedforward of data enables mid-course analysis and schedule adjustments in a planned manner, rather than in

a reactive, fire fighting mode. Data collected over a four-year period enabled those managing similar projects to estimate from previous project history and to compare current performance to past experience. By recognizing the pervasiveness of defects in software development and taking steps to count and measure the impacts of defects on schedules and productivity, project managers gained a better understanding of their projects and ultimately manufactured a better product in less time.

RiskExpert is a client/server expert systems tool being developed by the University of California, Berkeley to analyze the metrics [CHE94]. The METRIX tool developed by Gayet [GAY94] uses the optimized set reduction modeling technique to predict various phases of the software development process and improve software quality.

In the rest of this chapter, we describe our selected set of test and evaluation metrics in detail, and provide examples on how they can be utilized for various management level decisions.

2.4 Management Metrics

Management metrics are required by management to control the costs and schedules of a project. We identify and describe the following management metrics in this section:

- Cost metric
- Schedule metric
- Computer Resource Utilization metric
- Software Engineering Environment metric

2.4.1 Cost Metric

The cost metric tracks software development expenditures, and provides insight into the cost of the software development. Both estimated and actual costs are tracked for the following life cycle phases and categories of expenditures:

- requirement analysis
- design
- coding
- unit testing
- integration and testing
- formal qualification testing
- software problem/change resolution
- software engineering management
- software quality assurance
- software configuration management
- verification and validation
- tools
- new equipment.

The cost metric is the sum of all the above costs.

Data Elements / Attributes (A_i)	Domains (D_i)	Description
Data_Date	*<Year>/<Month>/<Day>* Subdomains: *<Year>*:0000..1999 *<Month>*:01..12 *<Day>*:01..31	The date the data elements were collected.
System_Name	Character	Name of the Computer Software Configuration Item
Activity_Type	"Requirements Analysis", "Design", "Code and Unit Testing", "CSC Integration and Testing", "Formal Qualification Testing", "Software Problem Change Report Resolution", "CSCI Integration and Testing", "Software Engineering Management", "Software Quality Assurance", "Software Configuration Management", "Verification and Validation", "Tools, "New Equipment and Facilities", "Software Data and Project Totals".	This is the service or product associated with the collected data. The Activity_Type varies depending upon whether the data is collected at CSCI or System.
BCWS	0..999999999.99	Budgeted Cost of Work Scheduled
BCWP	0..999999999.99	Budgeted Cost of Work Performed
ACWP	0..999999999.99	Actual Cost of Work Performed

Table 2.1: Elements of the Cost Metric

The attributes for the Cost metric as illustrated in Table 2.1 are *<Data_Date, System_Name, Activity_Type, BWCS, BWCP, AWCP>*. To illustrate the evolutionary nature of software metrics, suppose as part of the cost metric, we may want to collect metrics denoting the cost of manpower for each activity in addition to software costs, so we add a new attribute as follows:

<Data_Date, System_Name, Activity_Type, BWCS, BWCP, AWCP, Manpower>

An additional domain D_{n+k} will also be added to the *Manpower* attribute of the cost metric consisting of the value range 0..999999999.99. Since we have

$$\alpha_k : Ç \rightarrow Ç \times D_{n+k}$$

where D_{n+k} is the new domain for the *Manpower* attribute, the current state of the metrics database will include the new *Manpower* attribute for the cost metric, i.e.

$$R_k = Ç \times D_{n+k} \rightarrow Ç$$

2.4.2 Schedule Metric

The schedule metric tracks projected versus actual progress, and indicates changes and adherence to the planned schedules. The schedule is measured in terms of periodic and cumulative delays over the milestones of the software development. Both planned and actual schedules for major milestones as well as key software deliverables should be tracked as they change over time.

The schedule metrics, when plotted for both planned and actual schedules as they change over time, provide indications of problems in meeting key events of deliverables. The higher the slope of the trend line for each event, the more problems are being encountered. Milestone slippages should be investigated. Potential clustering of key events should be guarded against.

The schedule metric can be used in conjunction with other metrics to judge program risk. For example, it can be used with the test coverage metrics to determine if there is enough time remaining on the current schedule to allow for the completion of all testing.

The Schedule metric can be hierarchically classified into two attribute levels. At the top level, the Schedule metric comprises of a set of *Activities*, each of which has a defined milestone. Thus:

Sc : <*Activity*>

where *Activity* is in the domain {"*CSCI Integration and Testing*", "*Software Engineering Management*", "*Software Quality Assurance*", "*Software Configuration Management*", "*Verification and Validation*", "*Tools*", "*New Equipment and Facilities*", "*Software Data*"}. New values may be added to the *Activity* domain or existing domain values may be modified according to the needs of different organizations. The *Activity* attribute comprises further attributes each of which have different domains. The attributes for the each of the activities of the Schedule metric as illustrated in Table 2.2 are

<Data_Date, System_Name, Event_Name, Plan_Start_Date, Plan_End_Date, Actual_Start_Date, Actual_End_Date>

Data Elements / Attributes (A_i)	Domains (D_i)	Description
Data_Date	YYYY/MM/DD <*Year*>/<*Month*>/<*Day*> Subdomains: <*Year*>:0000..1999 <*Month*>:01..12 <*Day*>:01..31	The date the data elements were collected.
System_Name	Character	Name of the Computer Software Configuration Item
Event_Name	"Formal System Review", "Testing Events", "Software Data Product Deliveries" (dependent on organization)	This is the name of the milestone, deliverable, event or activity this record describes
Plan_Start_Date	As for Data_Date	The date on which the event is planned to start
Plan_End_Date	As for Data_Date	The date on which the event is planned to be completed
Actual_Start_Date	As for Data_Date	The date on which the event actually starts
Actual_End_Date	As for Data_Date	The date on which the event actually ends

Table 2.2: Elements of the Schedule Metric

Again, to illustrate the evolutionary nature of software metrics, suppose as part of the schedule metric, we may want to collect metrics denoting the percentage of task estimated to be completed for each activity at the current time. This will give us an

indication as to whether we can realistically expect to meet the planned schedule. We therefore add a new attribute *Percent_Complete* to the *Activity* domain as follows:

<Data_Date, System_Name, Event_Name, Plan_Start_Date, Plan_End_Date, Actual_Start_Date, Actual_End_Date, Percent_Complete>.

An additional second-level domain D_{n+k} will also be added to the *Activity* domain of the *Schedule* metric consisting of the domain 0..99. Since we have

$$\alpha_k : Ç \rightarrow Ç \times D_{n+k}$$

where D_{n+k} is the new domain for the *Percent_Complete* attribute for the Activity attribute of the Schedule metric, the current state of the metrics database will include the new attribute, i.e.

$$R_k = Ç \times D_{n+k} \rightarrow Ç$$

In general, we can further subdivide attributes to comprise non-terminal upper-level attributes (which have no domains) and terminal upper-level attributes (which have associated domains). Non-terminal upper-level attributes comprise a set of lower-level attributes, and alternatively represent the attribute A_i with domains D_i with

$$D_i = D_{i1} \times D_{i2} \times D_{i3} \ldots . \times D_{ik}$$

where D_i is the upper-level attribute and D_{ij}, j=1,..,k are the lower-level attributes.

2.4.3 Computer Resource Utilization Metric

The computer resource utilization metric tracks projected versus actual computer resource usage. It shows the degree to which estimates and measurements of the target computer resources such as CPU, memory, I/O, disk, and network bandwidth used are changing or approaching the limits of resource availability and specified constraints. Over-utilization of the resources has an impact on the cost and schedule, so this metric is

in some sense correlated with the cost and schedule metrics. The quantitative measurements of CPU utilization, memory utilization, I/O bandwidth utilization, disk utilization, network bandwidth utilization are easily achievable through known performance analysis tools. Approaching resource capacity may necessitate hardware change or software redesign. Exceeding specified reserve requirements can have similar impacts in the post-deployment phase. Proper use of this metric can also assure that each processor in the system has adequate reserve to allow for future growth due to changing or additional requirements without requiring re-design.

Early in the design phase, CPU utilization budgets should be established for each processor in the system. I/O utilization and throughput budgets should be allocated to each I/O channel in the system. Actual memory and I/O usage should be measured monthly during coding, unit testing, integration testing, and system-level testing. Measurements of CPU usage should also be made at regular intervals after the beginning of unit testing. Actual utilization should be formally demonstrated at the system level for each resource under peak loading conditions.

Figure 2.2 illustrates a sample CPU utilization graph for the Computer Resource Utilization metric over time. The illustration shows the target upper bound utilization as a straight line. In reality, the target upper bound utilization can change over time. The allocation for each resource type should not exceed the target upper bound utilization for any category. The figure shows zero CPU resource utilization in the first three months of the project, possibly due to the fact that such resources are seldom used in the requirements and design phases of the project. CPU utilization in the initial phases is well below the projected utilization. As the project progresses to the coding phases, more CPU

resources are utilized. This gradually increases in the testing and subsequent phases until it is close to the projected utilization.

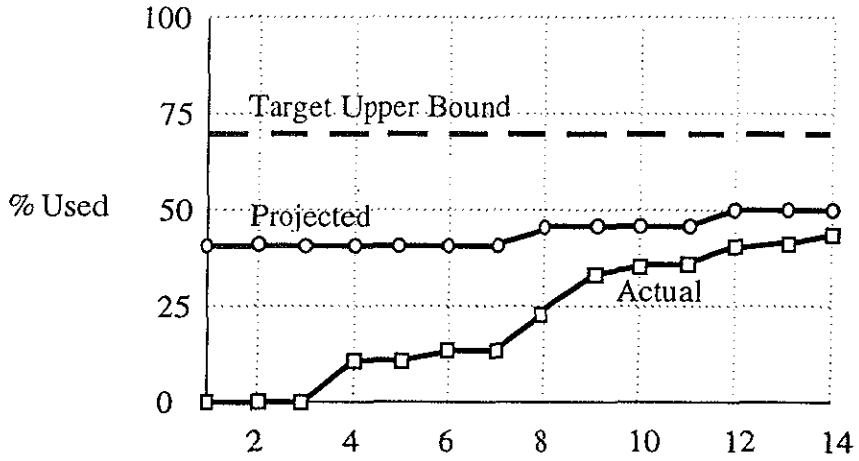

Figure 2.2: Sample CPU Utilization Graph for Computer Resource Utilization Metric

Resource utilization tends to increase over the development of a project. Therefore, adequate planning must be done up front to ensure that the software's operation does not put undue demands on the target hardware's capabilities. This measure allows one to track utilization over time to make sure that target upper bound utilization is not exceeded and that sufficient excess capacity remains for future growth and for periods of high stress loading. In instances where the development and target environments differ in types and/or capacities, caution must be taken in computing and analyzing the measures. Translations are acceptable up to a certain point, but testing on the target hardware must take place as early as possible. Initial estimates should be retained for comparison with what is finally achieved in order to aid in scoping future programs.

During development, it is important to look at both actual and projected values in relation to the target upper bound values. If either exceeds the target values, extra attention should be paid to assure that the projections drop to below the target upper bound value by project completion. If it is apparent from the projections that the target upper bound limits will be exceeded, action must be taken to either optimize the software or upgrade the capability of the target configuration.

Sudden drops in utilization may reflect either new systems capacity of new and more efficient software. The computer resource utilization metrics should be used in conjunction with the test coverage/success metrics (breadth and depth of testing) to ensure that measures of the actual usage are representative and portray the entire system under realistic stress loads.

2.4.4 Software Engineering Environment Metric

The Software Engineering Environment metric is used to assess developer's process maturity. It rates the developer's applied software engineering principles. Examples of these principles are structured design, usage of software tools, use of the program design language, etc.

The Software Engineering Environment rating provides a consistent measure of the capability of a contractor to use modern software engineering techniques in his development process, and therefore his capability to instill such principles and characteristics in a product. The basic assumption to this approach is that a quality process results in a quality product. The other test and evaluation metrics discussed in this chapter should be used to examine the quality of the product.

The Software Engineering Environment rating should be used during the source selection process. Besides the use as a tool with which to compare relatively the ability of contractors, the use of the Software Engineering Environment rating may encourage contractors to improve their software development process in order to increase their rating. A higher rating will increase the contractor's chance of being selected for future software development projects.

2.5 Requirements Metrics

Requirements and specifications change several times during product development. These changes may be due to new application demands, technology changes and/or errors in assumptions or interpretations. These changes are made when the history of previous design decisions are forgotten or missing. The requirement changes could be quite sophisticated and they may require more effort to figure out the types of repairs that need to be made. Often, such changes provoke new errors because designers may not be fully aware of the early assumptions made on the code.

2.5.1 Requirement Traceability Metric

The Requirements Traceability metric measures variance from system specifications. It measures the adherence of the software products (including design and code) to the requirements specifications. It helps the user of the software to estimate the operational impact of the software problems.

Analysis of the requirements traceability metric is performed by the development of a software requirements traceability matrix (SRTM). The SRTM is the product of a structured, top-down hierarchical analysis that traces the software requirements through the design to the code and test documentation. The SRTM will be completed to various

degrees depending on the current stage of the software life cycle. From the SRTM, various statistics can be calculated indicating the percentage of tracing to various levels, such as:

- the percentage of software requirements in unit-level design
- the percentage of software requirements in system-level design
- the percentage of software requirements in code
- the percentage of software requirements having test cases for all of its modules

and so on.

The requirements traceability metric should be collected starting from the requirements definition phase of the software life cycle. By the nature of the software development process, especially in conjunction with an evolutionary development strategy, the tracing of requirements will be an iterative process. Therefore, as new software releases add more functionality to the system, the trace of requirements will have to be revisited and augmented.

Those modules which appear most often in the matrix are most crucial in that they are required for multiple functions or requirements, can be highlighted for earlier development and increased test scrutiny.

The requirements traceability metrics should be used in conjunction with the test coverage metrics (depth and breadth of testing) to verify if sufficient functionality has been demonstrated to warrant proceeding to the next stage of development or testing. They should also be used in conjunction with the design stability and requirements stability metrics.

2.5.2 Requirement Stability Metric

The requirements stability metric measures requirement changes over time. It indicates the degree to which changes in software requirements affects the development effort of the software. It also allows for determining the cause of requirements changes. The metric is computed depending on two factors:

- the number of requirement changes on a periodic and cumulative basis, and
- the number of modules and lines of codes changed due to these requirement changes.

When a program is begun, the details of its operation and design are rarely complete, so it is normal to experience changes in the specifications as the requirements become better defined over time. When design reviews reveal inconsistencies, a discrepancy report is generated. Closure is accomplished by modifying the design or the requirements. The plot of open discrepancies can be expected to spike upward at each review and to diminish thereafter as the discrepancies are closed out. Good requirements stability is indicated by a leveling off of the cumulative discrepancies curve with most discrepancies having reached closure. A sample graphical display of the design stability metric is illustrated in Figure 2.3.

Causes of program turbulence can be investigated by looking at requirements stability and design stability together. If design stability is low and requirements stability is high, the designer/coder interface is suspect. If design stability is high and requirements stability is low, the interface between the user and the design activity is suspect. If both design stability and requirements stability are low, both the interfaces between the design activity and the code activity and between the user and the design

activity is suspect. The metrics for requirements stability should also be used in conjunction with those for requirements traceability and fault profiles.

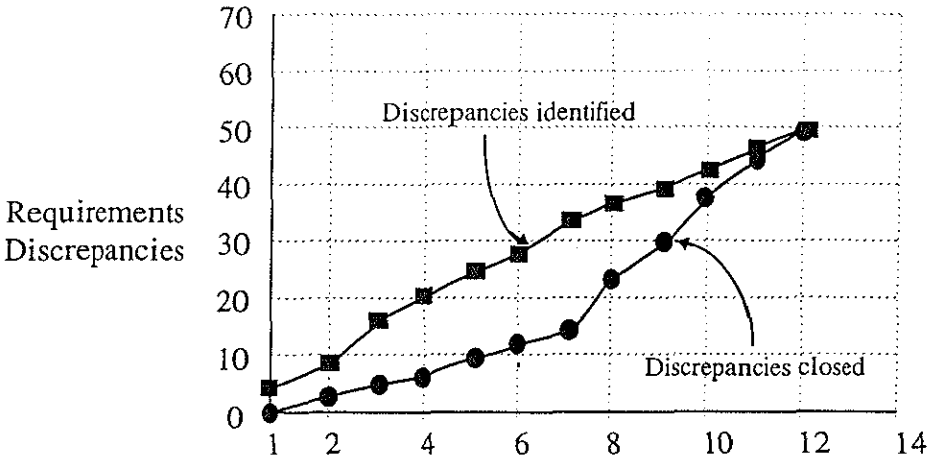

Figure 2.3: Graphical Display of Design Stability Metric

Allowance should be made for higher instability in the case where rapid prototyping is utilized. At some point in the development effort, the requirement should firm so that only design and implementation issues will cause further changes to the specification.

2.6 Quality Metrics

The quality of software products cannot be measured quantitatively or objectively. Factors that define quality include availability, dependability, user friendliness, and maintainability. In this section, we discuss the use of the following software metrics to measure software quality:

- Design Stability metric

- Complexity metric
- Breadth of Testing metric
- Depth of Testing metric
- Fault Profiles metric
- Reliability metric.

2.6.1 Design Stability Metric

The design stability metric illustrates software design changes over time. Design stability is used to indicate the amount of changes made to the design of the software. The design progress ratio show how the completeness of the design is advancing over time and helps give an indication of how to view the stability in relation to the total projected design.

Design stability should be monitored to determine the number and potential impact of design changes, additions, and deletions on the software configuration. The trend of design stability over time and releases provides an indication of whether the software design is approaching a stable state, that is, a leveling off of the curve at a value close to or equal to one. In addition to a high value and level curve the following other characteristics of the software should be exhibited:

- requirements stability is high
- depth of testing is high
- the fault profile curve has leveled off and most software trouble reports have been closed.

The higher the stability, the better the chances of a stable software configuration. Allowances for exceptional behavior of this metric should be made for the use of rapid

prototyping. It is thought that rapid prototyping, while possibly causing lower stability numbers (ie higher instability) early in the program, will positively affect the stability metric during later stages of development.

The design stability metric can be used in conjunction with the complexity metric to highlight changes to the most complex modules. It can also be used with the requirements metrics to highlight changes to modules which support the most critical user requirements.

2.6.2 Complexity Metric

The complexity metric evaluates and measures the structure of modules. Complexity measures give an indication of the structure of the software and provide a means to measure, quantify and/or evaluate the structure of software modules. They also indicate the degree of unit testing which needs to be performed. This metric also indicates the degree of unit testing which needs to be performed. It is commonly believed that the more complex a piece of software is, the harder it is to test and maintain the software. Additionally, lower complexity ratings reflect software that is easier to test and maintain, thus logically resulting in fewer errors and lower life cycle costs.

The control flow of a program can be represented by a directed graph. The complexity of the program can then be computed from the cyclomatic number. Although the concept of cyclomatic numbers has been used in graph theory, it became a useful metric after McCabe applied it to compute a program's complexity. McCabe's cyclomatic complexity metric measures the number of decision statements in a program: Given any piece of software, a control flow graph can be drawn wherein each node corresponds to a block of sequential code and each arc corresponds to a branch or decision point.

A sample graphical display for the McCabe's cyclomatic complexity metric is illustrated in Figure 2.4. McCabe's cyclomatic complexity metric is described here:

Let

E = # of edges (program flows between nodes)

N = # of nodes (sequential groups of program statements)

P = # of connected components (on a flow graph, it is the number of disconnected parts)

Compute:

Cyclomatic Complexity: C = E - N + 2P

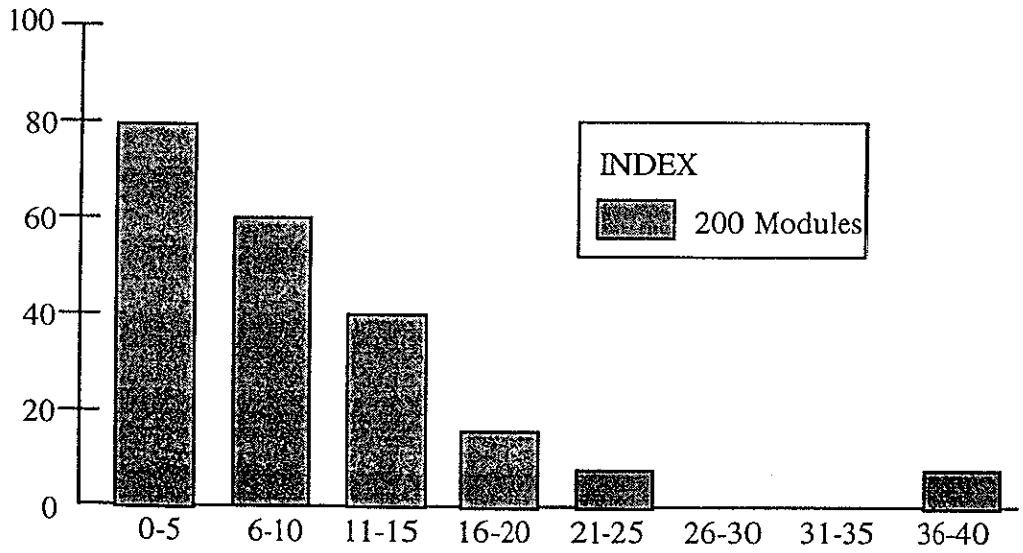

Figure 2.4: McCabe Cyclomatic Complexity Metric Example

The cyclomatic complexity has been related to programming effort, debugging performance, and maintenance effort. As a guideline, any module with cyclomatic

complexity greater than 10 is considered complex, and may need to be restructured, if feasible, into several less complex ones. It may also be advisable to limit the cyclomatic factor to 7 in the design phase to allow for expected growth to a value of 10 during implementation to code. McCabe's approach is appealing because it is simple to apply and can be computed for programs written in any programming language. The main drawbacks of the concept is that it only gives a somewhat superficial view of complexity since it does not give us measures such as the number of strongly connected subgraphs, their nestedness or loop structures, etc. While other loop estimation measures exist, they have not been as popular as the cyclomatic number based measures.

Halstead's metrics estimate a program's length and volume based on its vocabulary (operators and operands). Other simpler complexity metrics are control flow, the number of executable lines of code per module (which relates to the understandability of the module), and the percent of comment lines. There are additional ways of computing complexity. One such way is to calculate the number of control tokens + 1. Control tokens are programming language statements which in some way provide decision points which modify the top-down flow of the program. In other words, statements such as IF, GOTO, CASE, etc. are considered to be control tokens since they base program flow upon a logical decision thereby creating alternative paths which program execution may follow. A CASE statement would contribute (N-1) to complexity, where N is the number of conditions or cases associated with the statement.

Halstead's metric can be computed as follows:

Let n_1 = # distinct operators

n_2 = # distinct operands

N_1 = total # occurrences of the operators

N_2 = total # occurrences of the operands

Compute:

Vocabulary: $v = n_1 + n_2$

Program Length: $L = N_1 + N_2$

Volume: $V = L (\log_2 v)$

Some simpler complexity metrics are control flow, the number of executable lines of code per module (which relates to the understandability of the module), and the percent of comment lines. The executable lines of code metric can be computed by simply counting the number of executable lines of code in each module. Also, the percent comment lines metric can be calculated as follows:

Percent comment lines = (C / T) * 100

where

C = # comment lines in module

T = total # lines in module

The complexity metric is used throughout the software life cycle for each module in the system. Examination at various levels can provide indications of potential problem areas. Contractually limiting the complexity limit will simulate structured programming techniques, thereby impacting design by limiting the number of basis paths in a program at the design and coding stages. It is used during software testing to identify basis paths, to define and prioritize the testing effort, and to assess the completeness of module testing. During the maintenance phases, a proposed change should not be allowed to

substantially increase the complexity, thereby increasing the testing effort and decreasing maintainability.

Examination of complexity trends over time can also provide useful insights, especially when combined with other metrics such as design stability. For example, late software code "patches" may cause the complexity of the patched module to exceed an acceptable limit, indicating that the design rather than the code should have been changed. It is noted that the amount of testing is typically better judged by the relative structural complexity of the modules tested rather than by the sheer number of lines of code tested.

2.6.3 Breadth Of Testing Metric

This metric measures the degree of demonstrated functionality. Breadth of testing addresses the degree to which required functionality has been successfully demonstrated as well as the amount of testing that has been performed. This testing can be called *black box* testing. This metric is divided into three parts:

- coverage, representing the total number of requirements tested
- test success, representing the percentage of requirements passed, and
- overall success, representing the number of requirements passed with respect to the total number of requirements.

This is calculated as follows:

$$\frac{\#\text{ requirements tested}}{\text{total }\#\text{ requirements}} * \frac{\#\text{ requirements passed}}{\#\text{ requirements tested}} = \frac{\#\text{ requirements passed}}{\text{total }\#\text{ requirements}}$$

The coverage portion of breadth of testing indicates the amount of testing performed without regard to success. By observing the trend of coverage over time, one gets an idea of the extent of full testing that has been performed. The success portion of breadth of testing provides indications about requirements that have been successfully demonstrated during testing. By observing the trend of the overall success portion of breadth of testing over time, one gets an idea of the growth in successfully demonstrated functionality.

The breadth of testing metrics for coverage and overall success should be used together and in conjunction with the requirements traceability metrics, and fault profiles so that potential problem areas can be identified. The breadth of testing metric must be used in conjunction with the metrics for depth of testing, requirements stability, and design stability.

2.6.4 Depth Of Testing Metric

The Depth of Testing metric provide indications of the extent and success of white box testing. Depth of testing consists of four separate measures, each of which is comprised of one coverage and two success sub-elements.

The Depth of Testing metrics should be used in conjunction with requirements traceability, fault profiles, and complexity. For example, with complexity, the modules of highest complexity could be highlighted for testing. They must be used with the breadth of testing metrics to ensure that all aspects of testing are approaching an acceptable state for the user.

2.6.5 Fault Profiles Metric

The Fault Profiles metric tracks open and closed software trouble reports. Fault profiles provide insight into the quality of the software, as well as the engineer's ability to

fix known faults. These insights actually come from measuring the lack of quality (i.e. faults) in the software. Early in the development process, fault profiles can be used to measure the quality of the translation of the software requirements into the design. Later, they can be used to measure the quality of the implementation of the software requirements into design and then code.

There are various aspects of fault profiles that can be examined for insights into quality problems. The most popular type of graphical representation displays detected faults and closed (corrected and verified) faults on the same scale. These types of graphs should be examined for each priority level, and for each major module in the project. Applied during the early stages of development, fault profiles measure the quality of the translation of the software requirements into the design. Bugs reported during this phase suggest that requirements are not being defined correctly, completely, or at all. Applied later in the development process, assuming adequate testing, fault profiles measure the implementation of the requirements and design into code. Bugs reported during this stage could be the result of having incorrect requirements or an inadequate design to implement those requirements.

Fault profiles can also be used to show open and/or closed defects by type, by priority, by open age, by development phase, etc. Caution must be used in interpreting the fault profiles, as the detection of errors is closely tied to the quality of the development and testing process. That is, a low number of detected faults could indicate a good product from a good process or simply a bad process to start with, such as one with inadequate testing. A large number of bugs reported in a particular month may be the result of errors detected during a review of the specifications, audit, test, or from use of

the software in the field. Thus, the measures cannot be assessed without also considering the measures on breadth and depth of testing. The fault profiles should also be used in conjunction with the metrics for complexity, design stability, and requirements stability.

If the cumulative number of closed software bug reports remains constant over time and a number of software bug reports remain open, this may indicate a lack of problem resolution. The age of the open software bug reports should be checked to see if they have been open for an unreasonable period of time. If so, these software bug reports represent areas of increased risk. The cause for lack of resolution need to be identified and corrective action taken. Furthermore, once the average age of a software bug report has been established, large individual deviations should be investigated. There are several reasons why software bug reports may remain open for a lengthy period of time. One could be that the software bug report is a result of identification of an inadequate requirement which needs to be refined and is undergoing review. It could also mean that the responsible engineer has failed to take corrective action on the problem. Again, the reasons for lack of problem resolution need to be identified and corrective action taken.

2.6.6 Reliability Metric

The Reliability metric assesses the degree of completeness of development efforts. It is an indicator of continuous operation and how many faults there are in the software as well as the number of faults expected when the software is used in its most stressful intended environment.

This metric is the most controversial measure within this set, primarily because of the lack of a consensus opinion about what constitutes software reliability. Measuring the reliability of software is universally embraced as being important, but there are numerous

software reliability models, many of them very complex. There are also techniques such as fault seeding, which attempts to identify latent software bugs, and mutation testing, which attempts to assess the sufficiency of test cases. These techniques are not used in our work, primarily due to the labor intensive effort they require.

The algorithm proposed for the reliability metric uses fault profile metrics throughout development. The fault profile metrics indicates the rate at which faults are being reduced and thus reliability increased. The test coverage metrics should also be simultaneously considered. The Mean Time Between Failure (MTBF) measure estimates how often one can expect the software to "fail" in a field environment as long as inputs are of the type and in relative proportion to what will be encountered in field use and modules are exercised with the relative frequency expected during field use.

2.7 Relationships Among Metrics

The test and evaluation metrics thus identified can be used on their own with regard to other metrics to analyze their behavior over time. In addition, interrelated metrics can be more usefully analyzed with each other to determine the effect that one metric has on another metric. Relationships between metrics can be viewed more easily in the form of a two-dimensional matrix, with each row and each column representing one metric. This is illustrated in Figure 2.5. The matrix is symmetric: an 'X' in the row or column between two metrics indicates that the two metrics are typically used together under some situations as illustrated later in this chapter and also in subsequent chapters. For example, the 'X' in the row and column between the Schedule and Depth of Testing metric indicates that the Schedule metric can be used together with the Depth of Testing metric to determine if there is enough time remaining on the current schedule to allow for

completion of all testing. A 'X' in more than one row of a metric indicates that the metric is used together with several other metrics.

	Co	Sc	Cr	Se	Rt	Rs	Ds	Cm	Bt	Dt	Fp	Re
Co		X	X									
Sc	X								X	X	X	
Cr	X											
Se							X	X	X	X	X	X
Rt						X	X		X	X		
Rs					X		X		X			
Ds				X	X	X		X	X	X	X	
Cm				X			X		X		X	X
Bt		X		X	X	X	X			X	X	X
Dt		X		X	X		X	X	X		X	X
Fp		X		X		X	X	X	X	X		X
Re				X				X	X	X	X	

Figure 2.5: Relationships Among Test and Evaluation Metrics

Note that the diagonal matrix elements are invalid as both row and column refer to the same metric. The matrix is not static; Figure 2.5 only illustrates the relationships currently identified among various metrics at this time. Additional relationships identifed at a later time or by different organizations can easily be added to the matrix as the metric

database evolves. A brief explanation of each of the elements of the matrix is given below:

- the Cost metric may be used in conjunction with the Schedule metric and the Computer Resource Utilization metric. Using the cost metric with the schedule metric can determine how much of the budgeted amount has been spent relative to the milestone reached, and can provide an indication as to whether project costs may be underbudgeted relative to schedule. When used with the Computer Resource Utilization metric, the Cost metric can provide an indication as to whether there is sufficient money in the budget to purchase additional hardware resources as needs rise. Since resource utilization tends to increase over the development of a project, high computer resource utilizations midway through a project may imply the need for a budget to purchase additional hardware.
- in addition to being used in conjunction with the Cost metric, the Schedule metric is used together not only with the Depth of Testing metric, but also with the Breadth of Testing and Fault Profiles metrics (collectively referred to as the test coverage metrics) to determine if there is enough time on the schedule to allow for completion of all testing.
- the Software Engineering Environment metric should be used together with the Quality metrics to encourage contractors to improve their software development process in order to increase their rating, thus increasing their chances of being selected for future projects. For example, a high rating for the Software Engineering Environment metric would imply low complexity, high reliability, and more comprehensive coverage for the Breadth of Testing and Depth of Testing metrics.

- the Requirements Traceability metrics should be used in conjunction with the Depth of Testing and Breadth of Testing metrics to verify that sufficient functionality has been demonstrated. They should also be used in conjunction with the design stability and requirements traceability metrics.
- the Requirements Stability metric should be used in conjunction with those for requirements traceability and fault profiles. Unstable and constantly changing requirements signal potential problems for the project and can result in a high number of faults. Furthermore, it will be difficult to project managers to trace design and code to requirements if the project requirements constantly change.
- similar arguments hold for the other metrics, as discussed in the previous section.

2.8 Conceptual Modeling of Software Metrics

The relationship among the different software metrics and the management process in general and the use of metrics in software development can also be illustrated by a conceptual model such as that given in Figure 2.6 [MER96]. An explanation of the terminology used in the conceptual model is as follows:

1. **Project** is decomposed into **Phases**. Each **Phase** has an expected start date, expected end date, actual start date and actual end date. **Phases** may overlap in time.
2. **Phases** are decomposed into level 1 **Activities**. If necessary, level 1 **Activities** can decomposed into level 2 **Activities**, etc. Like **Phases**, each **Activity** has an expected start date, expected end date, actual start date and actual end date. **Activities** can overlap in time.
3. **Phases** and **Activities** have **Inputs** and **Deliverables**. **Inputs** are documents or other material that are required by a Phase/Activity. **Deliverables** are outputs produced by

Phase/Activity. An **Input** and **Deliverable** may be a **Document**, **Information**, **Experience**, etc.

4. Dependencies and relationships of various nature exist between **Phases** and **Activities**. Many of them are of vital interest to project managers and developers. For example, time dependencies (we cannot start **Activity** A1 before we completed **Activity** A2); input-output dependency (**Activity** A1 produces a **Delivarable** that is an **Input** to **Activity** A2); all **Activities** that have to do with a given **Requirement** or **DesignDecision** or **Module**.
5. Execution of **Phases** and **Activities** is governed by **Rules**. **Rules** reflect timing and other dependencies between **Phases/Activities** and take into account **Decisions** made by managers in various points of the project.
6. A **Milestone** is an important point in development. **Milestones** mark completion of important **Phases**. Each **Milestone** has a date when we expect to reach a **Milestone** and a list of things to be done when a **Milestone** is reached.
7. **Project Schedule** assigns expected start and end dates for **Phases** and **Activities**.
8. **Project Budget** is estimated **Cost** of **Phases** and **Activities**.
9. **Resources** are human **Effort** (measured in man-months), **Computer** resources (computer time + new equipment), investment in **Software** purchased for the **Project**, etc. **Cost** is a total cost of various **Resources** allocated to **Project**. Estimated **Cost** for the **Project** is computed based on estimated **Cost** for **Phases/Activities**.
10. A software **Project** consists of one or more **Modules**, each of which may comprise one or more **Routines**. A software **Product** is the final **Deliverable** from software

Project. **Bugs** occurring in a given **Module** are identified in a given **DevelopmentTask** and documented in a specified **Document**.

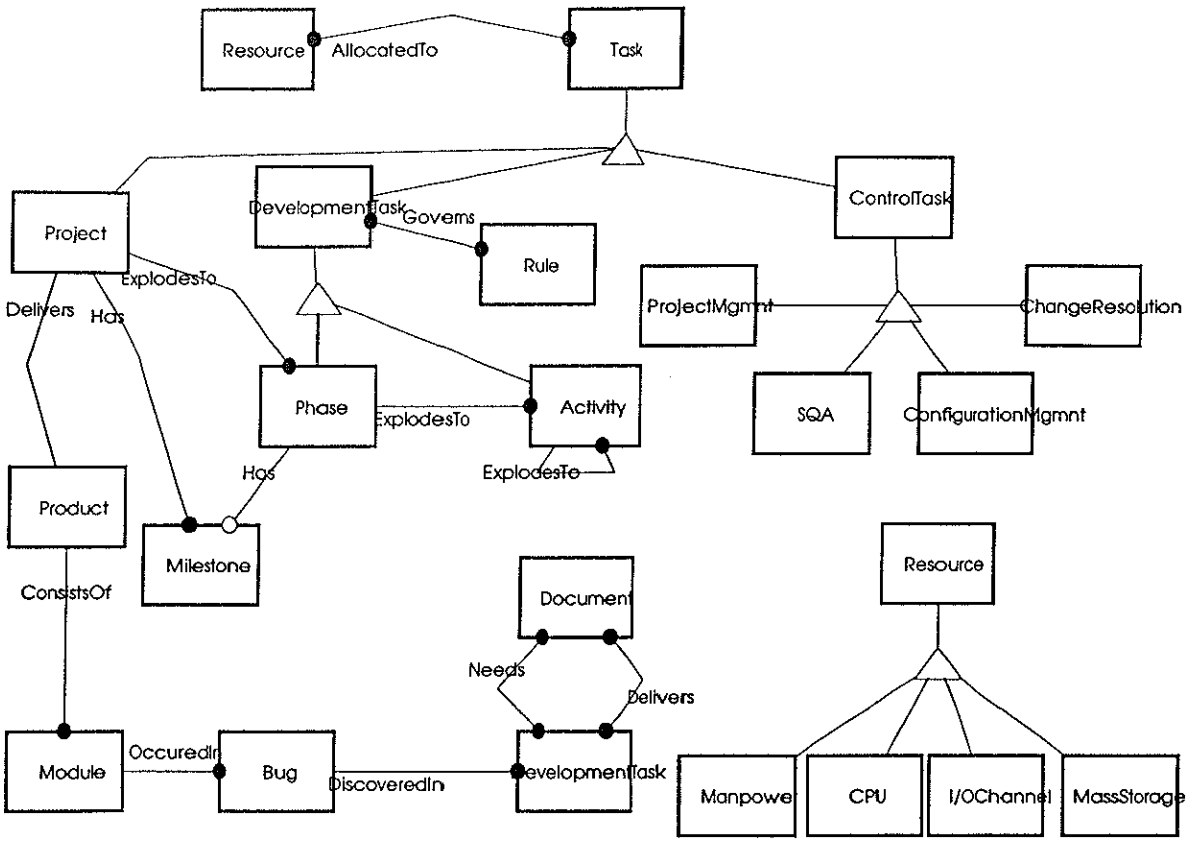

Figure 2.6: Conceptual Model of Use of Metrics In Software Development

Our notation is based on OMT [RUM91]. In the conceptual model, modeling concepts, called project entities, are in boxes. Triangle is *IsA* relationship, bold dot is '*many*' connector, circle is optional connector. Relationships for parent entities apply to derived entities. Derived entities inherit attributes from parents.

The conceptual model of the use of metrics at different phases of the project is given in Figure 2.7. In the model, a **Phase** of the project can be subdivided into **Planning**, **Requirements Analysis**, **Preliminary Design**, **Detail Design**, **Coding**, **Software Integration**, and **Releasing**. The **Requirements Analysis** phase can be further subdivided into **User Requirements Specification**, **Software Requirements**

Specification, and **Software Systems Specification**. The remaining parts of the conceptual model follow the same principles.

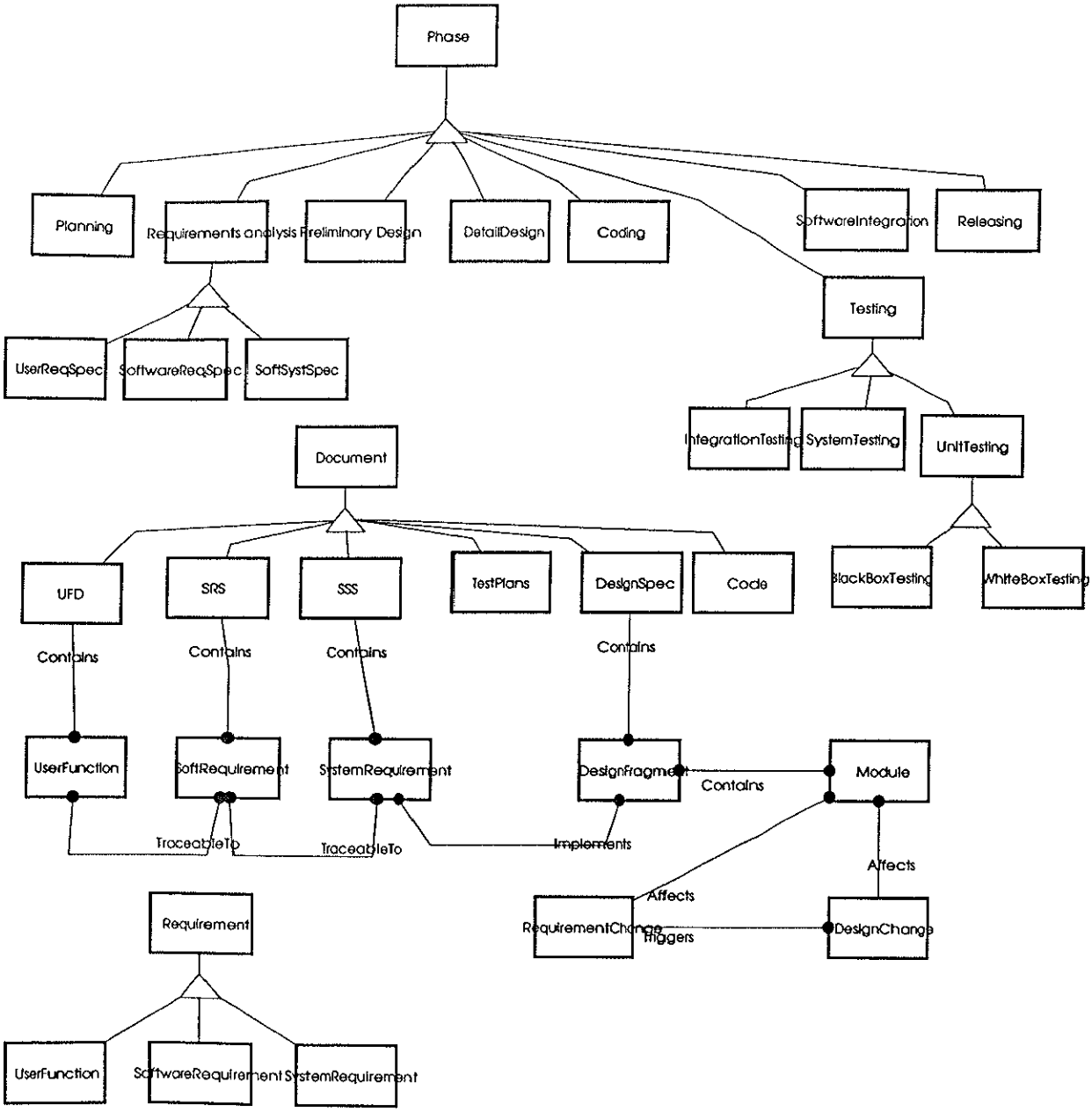

Figure 2.7: Conceptual Model of Components of Software Life Cycle

When applied to the entire organization, the collection of test and evaluation metric data will result in consistent data collection across all projects, providing a common base for measurement for related projects. Expert systems can then be used to

analyze the test and evaluation metrics for the current project and provide a decision aid to management. Advice and predictions can be given based on data from current project metrics and knowledge accumulated for previous projects in the metrics database. Based on this advice, management can assess the quality of the product and process by comparing expected project performance against the project plan, and revise plans accordingly and adjust expectations. The judicious use of the test and evaluation metrics can therefore aid in the management of software development. This process is illustrated in Figure 2.8.

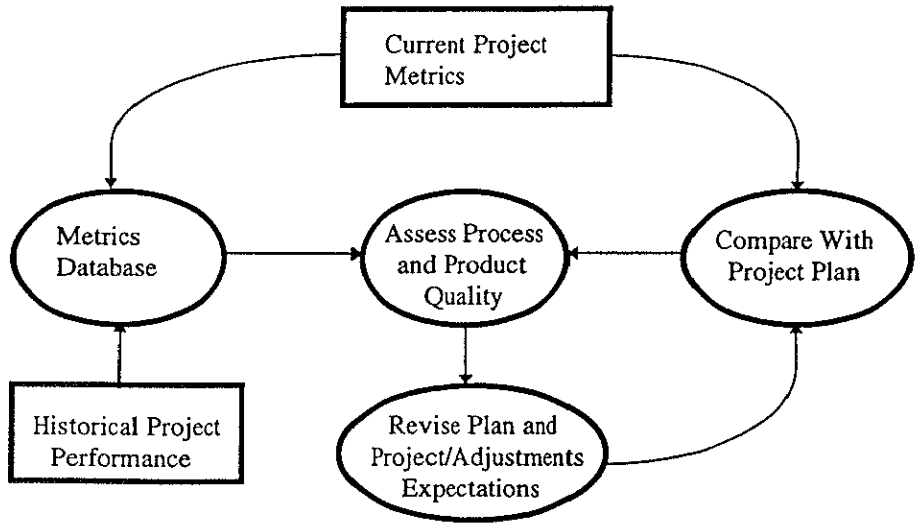

Figure 2.8: Management Through Measurement

In summary, the test and evaluation metrics, when properly used, will address the following:

- Identification of common risks associated with development
- Identification of questions or queries that can characterize these common risks
- Isolation of the risks thus identified

- Provision of opportunities for the manager to identify his own risks.

2.9 Sample Data for Software Metrics

We have just discussed the type of metrics to be collected and input to the metrics database. In Appendix A we provide sample data for several metrics. In the next chapter, we shall discuss some sample queries that can be made on these metrics, as well as more sophisticated metrics analysis techniques.