Product Models

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

Product models and metrics can be used to evaluate the process and product, estimate the cost and quality of the product, and as a feedback mechanism during software development and evolution as a monitor of the stability and quality of the product.

They provide a quantitative view of the software development process and product, can be used to refine and engineer techniques and tools, can help with technology transfer, and can be used to provide quality assurance probes into the product to provide visibility to anyone who needs it. In there most refined form they can be used in contracts for award, acceptance, and budget incentives.

In the next sections we will cover models and metrics for size, control and data structure complexity, reliability, and test coverage.

Size Models

Although size is the most common metric, it is the most difficult to articulate. That is because it depends upon so many different factors, e.g., the notation, the formatting style of the notation, what aspects of the product are considered part of its size (for example are comments extraneous information or essential parts of the size of a product), etc. For example, if we are interested in the size of a requirements document, a great deal depends upon whether the document is written in English or some formal notation and how we distinguish and enumerate requirements. Of the requirements are written in English and we are unable to distinguish or enumerate independent requirements then how do we model size? We could count pages of the document, which has been done, but now we are concerned about type font, spacing, etc.

The problems associated with models of size fall into two categories. The first is what do we count, i.e. what models for size do we use. These problems are partly addressed by the GQM in that the appropriate models to choose depend upon our goals. Deciding what we want to count, i.e. enumerated requirements, pages, etc. fall into this category. The second problem is associated with whether or not there are reasonable measures that can be associated with the model. Being able to distinguish requirements, i.e., having a mechanism for enumerating requirements falls into this category.

This problem is dependent upon our ability to formalize the notation of the product in such a way that the measure makes sense.

Since size metrics depend upon the product and its model, let us give some examples of size metrics and the products they can be used on.

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

None of these measures are good or bad in their own right. It depends upon how we use them, whether they satisfy the concept we are trying to analyze.

The number of pages of documentation can be associated with any document. It is a very gross measure and highly dependent upon the notation and page format. However, if one is interested in comparing products of similar notations and format is serves as a simple gross measure of size.

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

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

The number of functions in the specification is similar to the number of requirements except it is viewing functionality from the point of view of the designer, rather than the point of view of the user or customer.

The number of subsystems is primarily a measure of the design, as is the number of modules. However, both these definitions require careful definitions of the terms. As we have seen in the literature, module can be used to mean everything from a FORTRAN subroutine to an information hiding subsystem.

The number of function points is a derived measure associated with the amount of data being processed.

The number of procedures, functions or packages is a macro measure of code size or a micro measure of design. It can be used for estimation of effort since one can categorize these units by application or type and associate a data base of effort a a detailed level of specificity.

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

The number of tokens is a micro measure of the number of units of information. For example, if we are counting the number of tokens in a program, we might count the number of operators and operands in the source code. If we are counting the number of tokens in a requirements document written in English, we might count the number of nouns and verbs. In the next section we will cover a theory of program measurement based upon token count as suggested by Halstead, called software science.

Software Science Metrics

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

n_1 = # unique or distinct <u>operators</u> in an implementation
n_2 = #unique or distinct <u>operands</u> in an implementation
N_1 = total usage of all operators
N_2 = total usage of all operands
$f_{1,j}$ = # occurrences of the jth most frequent operator, $j = 1, 2, . . n_1$

$f_{2,j}$ = # occurrences of the jth most frequent operand, $j = 1, 2, .. n_2$

The vocabulary n of the algorithm or program is defined as

$$n = n_1 + n_2$$

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

$$N = N_1 + N_2$$

and

$$N_1 = \sum_{j=1}^{n1} f_{1,j} \quad N_2 = \sum_{j=1}^{n2} f_{2,j} \quad N = \sum_{i=1}^{2} \sum_{j=1}^{ni} f_{ij}$$

To better understand the definitions and the counting mechanism for operators and operands consider the following example algorithm.

Example: Euclid's Algorithm

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

Operator Parameters: Greatest Common Divisor Algorithm

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

Operand Parameters: Greatest Common Divisor Algorithm

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

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

$$N\hat{} = n_1 \times \log_2 n_1 + n_2 \times \log_2 n_2$$

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

Another metric, **program volume**, represents the size of an implementation,

$$V = N \log_2 n$$

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

The metric **potential volume**,

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

is an attempt to approximate the minimal number of tokens necessary to specify the algorithm. It represents the minimal number of input and output parameters and operators needed for a specification. It is based upon the program volume equation with N defined as 2 operands, the function name and the bracketing operator, ( ), for the parameters, plus n_2* representing the number of operands in the program occurring only once each.

Using V and V*, one can represent the metric **program level**, which is the level of an implementation, i.e.,

$$L = V^*/V$$

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

We can approximate L with L^, by the formula $(2 \times n_2)/ (n1 \times N_2)$ and define a metric for **program difficulty** of implementing an algorithm in a particular languages as

$$D = 1/L$$

Programming effort, E, can then be defined as the volume times the difficulty.

$$E = V \times D = V/L = V_2/V^*$$

It has been suggested that E might be thought of as representing the effort required to comprehend an implementation rather than to produce it. One can think of E as a measure of program clarity.

Cyclomatic Complexity

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

$$v(G) = e - n + 2xp$$

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

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

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

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

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

Segment-Global Usage Pairs

In an attempt to evaluate the usage of global data for a program, Basili and Turner used a metric, to study the quality of the use of globals from the point of view that if a variable is declared global, it should be used by most of the sements to which it is available.

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

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

RUP = AUP/PUP

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

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

This metric is dependent upon the programming language and what kind of hierarchical data structuring it allows.

Data Bindings Metric

Myers has argued that modules should be formed in a manner which reduces coupling and increases strength. In an effort to measure the data coupling between modules, Basili and Turner used the concept of data bindings to capture the coupling relationship between program segments.

A segment-global-segment data binding (p,r,q) is an occurrence of the following:
(1) segment p modifies global variable r
(2) variable r is accessed by segment q
(3) p and q are different

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

Letting ADB, actual data bindings, represent the absolute number of true data bindings in the program, i.e., the true connectivity, and PDB, possible data bindings, represent the absolute number of potential data bindings given the programÆs global variables and their declared scope, they defined RDB, relative percentage of data bindings as

RDB = ADB/PDB

the normalized number of data bindings wit+in the program.

One problem with the data bindings metric is that is does not take into account indirect bindings, through parameters and the calling program. Thus implementations of an abstract data type may show no explicit bindings, even though the program elements are tightly bound.

Span

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

```
X := Y ;
Z := Y ;
X := Y æ
```

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

Using Span, we can define complexity as either

- the number of spans at any point (using either the maximum, average, median of that number as the actual value

OR

- the number of statements a variable must be remembered on the average, e.g., average span over all variables.

One potential variation of this metric is to perform a live/dead variable analysis and then define the complexity of a program to be proportional to # variables alive at any statement.

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

$$C(M) = \sum_{i=1}^{\text{number of statements}} \frac{n_i \times s(n_i)}{\text{number of statements}}$$

where n_i = the number of spans of size $S(n_i)$.

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

Analysis of the Metrics

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

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

Do measures like cyclomatic complexity and the software science metrics relate to effort and quality?

Does the correspondence increase with greater accuracy of data reporting?

How do these metrics compare with traditional size metrics such as number of source lines or executable statements?

How do these metrics relate to one another?

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

The data used in the analysis was commercial software. The application domain was satellite ground support systems consisting of 51,000 to 112,000 lines of FORTRAN source code. Ten to 61% of source code was modified from previous projects. The development effort ranged from 6900 to 22,300 man hours. The analysis focused on data from 7 projects, only newly developed modules, i.e., subroutines, functions, main procedures and block dataÆs.

Metrics studied included:

- Source lines of code
- Source lines of code excluding comments
- Executable statements
- Software science metrics
  - N : Length in operators and operands
  - V : Volume
  - V* : Potential volume
  - L : Program level
  - E : Effort
  - B : Bugs
- Cyclomatic complexity
- Cyclomatic complexity excluding compound decisions
  (referred to as cyclo_cmplx_2)
- Number of procedure and function calls
- Calls plus jumps
- Revisions (versions) of the source in the program library
- Number of changes to the source code

Figure 6.1 show the relationship of the metrics to actual effort. It should be noted that since the data comes from real projects, there is the potential for error in the reported effort data. However since project data is reported in two forms, a reliability check was performed and the data is rated by assurance of its accuracy, given as a percent.

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

Figure 6.1 METRICS' RELATION TO ACTUAL EFFORT

The data shows that there is some relation between the metrics and the effort data across all projects. However the relation improves with individual project data, validated data, and individual programmers data. Note that the Spearman correlations (R values) are all significant at P = 0.001 level.
Figure 6.2 shows the relationship between the metrics and the number of faults found in the modules. The number of program faults for a given module is the number of system changes that listed the module as affected by an error correction. In order to differentiate among the various faults, faults weighted by the effort expended. Weighted faults (W_flts) is a measure of the amount of effort spent isolating and fixing faults in module. Note that the Spearman correlations (R values) are all significant at P = 0.001, except for those marked with an *, which were significant at P = 0.05.

Note that when compared with faults, the relations low overall; with the number of revisions showing the strongest relationship. However, as before the relations improve with individual projects or programmers.

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

Figure 6.2 METRICS' RELATION TO PROGRAM FAULTS

Figure 6.3 show the relationships among the various complexity metrics. There were 1794 modules considered in this study and the Spearman R Values are all significant at P = 0.001.

Here one may notice that many of the size and complexity metrics correlate quite closely with one another. This is especially true for the various lines of code measures, cyclomatic complexity, and the software science metrics. Thus there is an indication that they may be measuring the same thing. It should be noted however, that revisions, and calls do not correlate as well with the other metrics or with each other.

Based upon performing this study in a commercial environment, the authors concluded that you could use commercially obtained data to validate software metrics but that validity checks and accuracy ratings are useful. The strongest effort correlations are derived with modules from individual

programmers or certain validated projects and the majority of effort correlations increase with the more reliable data.

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

Figure 6.3 RELATIONSHIPS AMONG THE METRICS

With regard to the ability of the metrics to predict effort or quality, the authors concluded that none of the metrics seems to satisfactorily explain effort or development faults and that neither software scienceÆs E metric, cyclomatic complexity nor source lines relates convincingly better with effort than the others. The number of revisions correlates with development faults better than either software scienceÆs B metric, E metric, cyclomatic complexity or source lines of code

They also concluded that many of the size and complexity measures relate well with each other, which might mean they are measuring the same thing.

Automatable Metrics

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

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

An autmatable metric is considered useful, if it is sensitive to externally observable differences in the development environment and the relative values correspond to some intuitive notion about characteristic differences in the environment. Thus version number, in the earlier study might be considered useful in that is it sensitive to observable difference, e.g., the number of defects found in the product, and it corresponds to the fact that as the number of defects increases, so does the version number. However, this usefulness of an automated metric must be checked for the environment in which it is being used.

Examples of automatable metrics include **program changes** [Dunsmore and Gannon], textual revision in the source code representing one conceptual change, and **job steps** [Basili and Reiter], the number of computer accesses during development or maintenance.

More specifically, program changes are defined as textual revisions in the source code of a module during the development period, such that one program change should represent one conceptual change to the program. A program change is defined as: one or more changes to a single statement, one or more statements inserted between existing statements, or a change to a single statement followed by the insertion of new statements. The following are not counted as program changes: the deletion of one or more existing statements, the insertion of standard output statements or special compiler-provided debugging directives, or the insertion of blank lines or comments, the revision of comments and reformatting without alteration of existing statements. Program changes have been shown to correlate well with defects in a particular environment [Gannon and Dunsmore]

Job steps are defined as the number of computer accesses representing a single programmer-oriented activity performed on the computer at the operating system command level. Examples include text

editing, module compilation, program compilation, link editing, and program execution. Job steps are basic to the development effort and involves nontrivial expenditures of computer or human resources and were show to correlate with human effort in a particular environment.

Measuring the effects of methods on products

In this section we discuss an empirical investigation, involving the replication of a software development effort under different development approaches, with automatable measures of the development process and developed product.

The goal of the study was to analyze the effect of a disciplined approach to software development with respect to the its effect on the process and product.

The experiment was performed at the University of Maryland using graduate and advanced undergraduate students on advanced elective courses. The task was to design, implement and test a small compiler, based on a given specification. The project size was between one and two staff months, approximately 1200 source lines of code. The programming language was a high level structured programming language with string processing capability. The computer was a UNIVAC 1108. Participants were familiar with the language and host system. They had all taken a course in basic compiler writing. They were aware of being monitored but not what was being measured or why.

The experimental design was a replicated project study. There wee nineteen replications of the basic experiment. It was a one by three design, with the treatment factor being the approach and three treatment levels. Seven three person teams (DT) used a disciplined methodology which consisted of an integrated set of techniques including top down design, a process design language, design and code reading, walk-throughs, and chief programmer teams. These methods were taught as part of the course so the subjects were novices in the methodology. Six three person teams (AT) used what was termed as an ad hoc approach, i.e., the software development proceeded in a manner of the subjects own choosing; there was no methodology taught as part of the course. Six single individuals (AI) also used an ad hoc approach so we were able to study the effects of team versus individual development.

A large set of dependent variable (automatable metrics) were observed and extraneous factors (project, language, schedule, tools, etc.) were held constant. Although there was no explicit randomization of the subjects performed, a check of subject factors, such as prerequisite grades, revealed nearly random distribution.

During the experiment, the activities of each group were automatically monitored via special processors and a data bank consisting of source code for every module compilation and test data for each program execution was created.

The study can be viewed as exploratory, i.e., was it possible to observe quantifiable differences in the product and process based upon the following general hypotheses:
1) the disciplined approach reduces the average cost and complexity of the process, and

2) the disciplined team should behave more like an individual programmer than a team in terms of the resulting product (conceptual integrity).

With regard to the automated metrics, these hypotheses translate into:
1) DT <= AI = AT
2) AI <= DT <= AT or AT <= DT <= AI

Non-parametric statistical test were used because it was not possible to make any assumptions about the underlying distributions. Because of the exploratory nature of the experiment, individual critical levels with a cut off at .20 were employed as opposed to an arbitrary significance level.

With regard to process aspects, the data for program changes and job steps are given in Figure 6.4. With regard to program changes, you should notice that the seven disciplined teams were in the group of eight that generated the fewest program changes and that there is a mix of ad hoc teams and individuals that is difficult to differentiate in the rest of the group. This supports the hypothesis that for program changes DT < AT = AI (with P = .003). We see similar results and support for the hypothesis from the job steps data, i.e. DT < AT = AI for job steps (with P = .0036) with simialr resluts all categories of job steps.

Actual Data (Ordered)

Program Changes	**Job Steps**
DT_4 = 111	DT_2 = 44
DT_7 = 114	DT_6 = 58
DT_2 = 120	DT_1 = 67
DT_3 = 136	DT_3 = 68
DT_6 = 159	DT_4 = 79
AI_6 = 187	AI_6 = 87
DT_1 = 223	DT_3 = 90
DT_5 = 251	DT_7 = 123
AI_3 = 270	AT_5 = 150
AI_2 = 281	AI_3 = 151
AT_6 = 287	AI_1 = 159
AT_1 = 301	AT_6 = 164
AI_4 = 316	AT_4 = 173
AT_4 = 394	AI_5 = 176
AT_5 = 493	AI_4 = 183
AI_5 = 525	AT_1 = 216
AI_1 = 539	AT_3 = 266
AT_3 = 554	AI_2 = 351
AT_2 = 1107	AT_2 = 372

Figure 6.4 Raw Data for Program Changes and Job Steps

Although non-paramteric statistics were used, it might be interesting to note that the average for program changes were DT = 159, AI = 353 and AT = 522 and for job steps were DT = 76, AI = 186, and AT = 244.

The results for product aspects were not so clear. The product metrics examined were: the number of modules (separate compilations), segments (procedures and functions), source lines of code with comments, executable statements, statement types by count (if, case, while), statement type by percentages, the average statements per segment, the average statement nesting level, the number of decisions, the number of tokens, the data scope counts (global, parameter, local), the data scope variable percentages (global, parameter, local), segment-glabal usage percentage, andsegment-gloabl-segment data bindings. For those variable that showed a distinction, Figure 6.5 gives the form of the distinction.

Aspect	**Outcome**
modules	
segments	AI < AT = DT
source lines of code	AI < DT < AT
executable statements	
statement types by count	
if	DT = AI < AT
case	
while	
statement types by percentages	
if	DT = AI < AT
case	
while	
average statements per segment	AT = DT < AI
average statement nesting level	
number of decisions	DT = AI < AT
number of tokens	
data scope counts	
global	AI < AT = DT
parameter	AI < AT = DT
local	
data scope variable percentages	
global	
parameter	AI < AT = DT
local	AT = DT < AI
segment-glabal usage percentage	
segment-global-segment data bindings	
actual	
possible	DT = AI < AT

Figure 6.5 Product Aspects

For product aspects that showed a difference, the results fell into four categories: AT < AT =DT, AT = DT < AI, AI = DT <AT or AI , ST < AT, that is either AT <= DT <= AI or AI <= DT <= AT. Thus none of the observed programming aspects contravene the basic assumptions for product, i.e., either the disciplined team looked like a single programmer or a disciplined team but the major difference were between the ad hoc team and the individuals.

In this experiment, four variations for cyclomatic complexity were defined:

1) SIMPRED-NCASE -- Simple predicates contribute 1 unit; CASE statements contribute 1 unit for each case label.
2) SIMPRED-LOGCASE -- Simple predicates contribute 1 unit; CASE statements contribute $\log_2(n)$ units, where n is the number of case labels.
3) COMPRED-NCASE -- Compound predicates contribute 1 unit; CASE statements contribute 1 unit for each case branch; multiple case labels on the same case branch are disregarded.
4) COMPRED-LOGCASE -- Compound predicates contribute 1 unit; CASE statements contribute $\log_2(n)$ units, where n is the number of case branches; multiple case labels on the same case branch are disregarded.

Since Cyclomatic complexity is meant to be a segment metric rather than a system metric, the segments were distributed based upon the frequency of occurrence of the segments with different cyclomatic complexities. Then histograms were generated based upon the absolute frequencies and the relative cumulative frequencies. Various quartiles were then chosen for comparison. It should be noted that the Sum of the cyclomatic complexity across segments yielded similar results to the decisions result and the number of segments exceeding cyclomatic complexity of 10 showed no significant results.

The results for these four variations of cyclomatic complexity were:
1) SIMPPRED NCASE DT = AT < AI
2) SIMPPRED LOGCASE DT < AT = AI
3) COMPRED NCASE DT = AT < AI
4) COMPRED LOGCASE DT < AT = AI
Thus when the case statement contributes n decisions, the disciplined team looks like the ad hoc team and has a lower set of segments in the higher cyclomatic complexity quartiles, but when the case statement contributes log n decisions, the disciplined team has a lower set of segments in the higher cyclomatic complexity quartiles than either of the ad hoc groups. This might imply that the disciplined team say the solution at a simpler level, because the could compose combinations of if statements into a case statement.

The authors drew the conclusion that the study results should be interpreted as strong evidence that the disciplined methodology reduces the cost and defects of the process but weaker evidence that the disciplined team acts like individuals. More importantly, the results show that one can quantitatively demonstrate the effectiveness of the approach.

Based upon this study, further analysis was performed by Basili and Hutchens posing the following questions relative tot he same data set:

Are there more useful syntactic complexity measures?
  Which measures are most effective in predicting error proneness?
  How do the measures relate to one another?
  Are there other measures that demonstrate the differences in the groups?
  What data measures can be used to predict error proneness?

In order to answer the first question they generated a family of complexity metrics based upon program structure:

$$c(p) = b \sum_{i=1}^{m} c(p_i) + f(n,l,t,s)$$

where c = complexity measure for program component p with subcomponents p_i, i=1, 2, . . ., m
  b > 1 is a nesting constant factor
  n = number of decision nodes in p
  l = nesting level of p in the segment
  t = the type of component (e.g., if, while, . . .)
  s = the structuring of the component (e.g., proper, not proper [with abnormal exits] . . .)
  f = a function selected depending on the intended use of the metric

An example metric from this family is

$$SC = \quad c(p) = 1.1 \sum_{i=1}^{m} c(p_i) + \begin{cases} 1 + \log_2(n+1) & \text{if } p \text{ proper} \\ 2 + 2\log_2(n+1) & \text{if } p \text{ is not proper} \end{cases}$$

This particular member was called structural complexity and was chosen because it correlated well with program changes. The study compared five metrics: structural complexity (SC), statement count (SC), cyclomatic complexity (CV), decisions statements (DS), and calls (CA). A correlation was performed between each of the metrics, the results are given in Figure 6.6. As can be seen from the matrix, all of the metrics correlate quite well with one another, except possibly for calls.

	ST	SC	CA	CV	DS
ST	1.0				
SC	.975	1.0			
CA	.845	.770	1.0		
CV	.879	.893	.747	1.0	

DS	.873	.939	.617	.832	1.0

Figure 6.6 Correlations between the metrics

The high correlations between the metrics causes multiple regression equations to be suspect. Of the 19 projectÆs regression models, forty percent found no second variable useful, fifty percent found minor improvement with a second variable, and 10% entered a second variable with a <u>negative</u> coefficient while doubling or tripling the first variableÆs coefficient (indicating a very unstable model).

In studying the relation between complexity and program changes, several observations were made. First individual differences appear to wash out the correlations when multiple individuals are used. This may account for the lack of results in many experiments. Second, there is a strong correlation between complexity and program changes when individuals have been isolated, e.g., R^2 values between .5 and .9 for 6 projects studied. Third, individual ability to cope with complexity appears to vary widely; the slopes of the fitted lines were .16 to .73.

Combining teams, individuals and methodologies, the proposed metric, SC, was more strongly correlated with program changes than cyclomatic complexity in 14 out of the 17 projects, and statement count in 9 out of the 17 projects.

In order to further the original study, the authors plotted program changes against each of the metrics. The idea was to determine the effect of each of the treatment groups on lowering the slopes, i.e, reducing the program change rate. The slopes of each of the three groups, DT, AI, and AT were analyzed with respect to looking for differences in the approaches. The results are given in Figure 6.7. Based upon the data it appears that the ad hoc individuals had a higher slope and thus made more changes for a given level of complexity.

Slopes

SC (.05)
- AI > AT (.132)
- AI > DT (.014)

ST (.05)
- AI > AT (.094)
- AI > DT (.014)

CV (.02)
- AI > AT (.180)
- AI > DT (.008)
- AT > DT (.074)

DS (.02)
    AI > AT (.026)
    AI > DT (.008)

Figure 6.7 Methodology comparison with respect to slope reduction

In order to determine if the methodology treatment generated more consistency among the teams, for each of the lines, r^2 was computed to measure the closeness to a standard. The results are given in Figure 6.8. The results imply that as expected, individuals are more consistent than teams, but that disciplined teams are more consistent than ad hoc teams.

r^2

SC (.03)
    AI > AT (.016)
    AI > DT (.180)
    DT > AT (.052)

ST (.10)
    AI > AT (.026)
    DT > AT (.034)

CV (.10)
    AI > AT (.016)
    AI > DT (.128)

DS (.10)
    AI > AT (.042)
    AI > DT (.180)
    DT > AT (.138)

Figure 6.8 Methodology comparison with respect to r^2

In comparing the metrics, SC and ST appear to do better than CV, DS or CA at explaining program changes. The results are show in Figure 6.10. However, it should be noted that SC is complicated to calculate and ST comes free with most compilers, hence statement count appears to be a good a metric as any in predicting program changes, and hopefully faults.

SC > CV (.167) 13
SC > DS (.063) 14
SC > CA (.019) 15

ST > CV (.019) 15

ST > CA (.063) 14

Figure 6.9 Comparison of Metrics

Since there appears to be little gain from combining metrics, further questions were asked regarding alternative metrics:

Can metrics be used to generate a view of system modularity?

What does this view of system modularity say about the system?

How does this view relate to the designerÆs view?

If data bindings are a measure of strength and coupling, it follows that modules should be characterized as a cluster of program segments, the cluster determined by the data bindings.
Belady and Evangelist tried usage data bindings in determining clusters of a large system. There are many clustering methods. It is not clear which is the right one or if any one is better than another.

Basili and Hutchens used data bindings to generate a view of system modularity. To do this they defined several classes of data bindings:

A **potential data binding** is an ordered triple (p,x, q) where p and q are components and x is a data variable in the scope of both p and q.

A **usage data binding** is a potential data binding where p and q use x (for reference or assignment).

A **feasible data binding** is a usage data binding where p assigns and q references x. This is the actual data binding definition used by Basili and Turner.

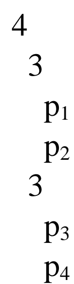
A **control flow data binding** is a feasible data binding where there is a possibility of control passing to q after p has had control.
The number of potential data bindings is the largest. As the definitions become more restrictive, smaller numbers of data bindings are found.

To deal with clustering, a dissimilarity matrix, D can be defined, such that $D_{ij} = D_{ji}$ = the percentage of bindings involving either i or j that also involves something other than i and j. Given this D, we may use the link cluster method, i.e., intuitively, the dissimilarity between any two elements should be directly related to the bindings involving them. Using the single link method, this is only true for the first iteration. They deal with this problem, they recalculated a dissimilarity matrix directly from the data at each iteration.

More formally, a dissimilarity matrix is an NxN, non-negative, real, symmetric matrix with zeros on the diagonal. It can be loosely thought of as a ôdistanceö matrix. For example, with data bindings: (p_1,a, p_2), (p_1, b, p_2), (p_3, c, p_4), (p_3, d, p_4), (p_2, e, p_3), and using $D_{ij} = 5 - B_{ij}$, i not equal to j, we get

$$D = \begin{matrix} 0 & 3 & 5 & 5 \\ 3 & 0 & 4 & 5 \\ 5 & 4 & 0 & 3 \\ 5 & 5 & 3 & 0 \end{matrix}$$

This allows you to produce a tree (called a dendogram) depicting the modularity with the leaves associated with the procedures:

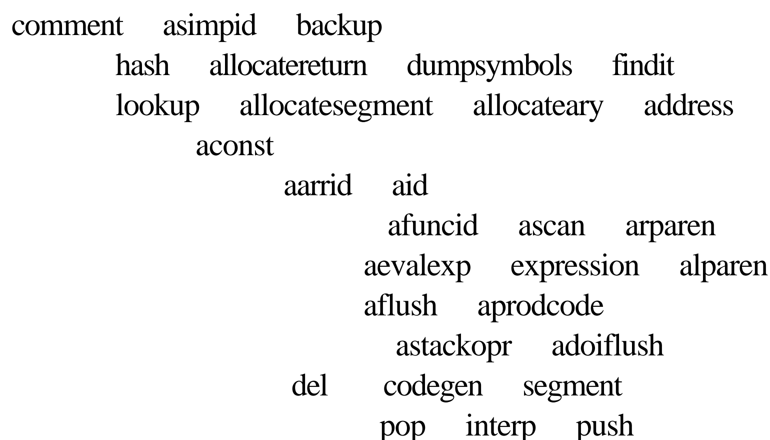


The way to interpret this dendogram is to note that p_1 and p_2 cluster together at the same level/pass that p_3 and p_4 cluster together.

To answer the questions posed about system modularity, two data sets were examined:
1. The compiler programs from the Basili/Reiter study

2. Two systems from the NASA/SEL data collection consisting of 65K and 100K lines of FORTRAN whose application is ground support software for satellites.

One dendogram produced from the Basili/Reiter data is:



program
stmt constget
nextchar scan mextsymp
specialchar identifier constant

The reader should notice that one of the most tightly clustered set of compnents is pop, interp, and push which are all parts of the code generator. Also nextchar, scan, nextsymb, specialchar, identifier, and constant are all parts of the scanning function. That is, the data binding clustering algorithm clearly clusetered components that were tightly connected with regard to the passing of data among them.

With regrad to the dendogram produced by the SEL data, the same phanomenon was observed. Note that the the system is decomposed into subsystems and each subsystem consists of components whose first two letters indicate the name of the subsystem to which they belong. The designers of the subsystem verified the fact that the dat bindings subsystem decomposition rep[resented their intuition of how the system should have been decomposed. A part of the dendogram produced by one of the SEL projects is:

```
.
.
.
HKGYROWT  HKMAGBAL  HKSUNBAL  HKSTRBAL  HKTIMCHK
HKFRMFIL
        HKSIMGYR
             HKGYRBAL
                  HKBYRAIR  HKGYROPR
        HKSIMMAG  HKMAGPRO  HKMAGMAC
        HKSTARPR  HKSIMSTA
        HKSUMPYS  HKSIMSUN  HKSUNPRO
        HKFSSPRO  HKSMMFSS
VRONLVER  VRRECORC  TPOBCVER
         VRGYRATE
               VRROTATE  VRGYREAD  VRMGORDR  VRMGREAD
                     VRMAGVER
                           VRSTNAVG  VRIRUVER
                      VRTMORDR
                           VRTMREAD  VRTIMVER
                       VRWHORDR  VRWHREAD  VRSRWVER
                       VRSUNCN2  VRSNREAD  VRSNORDR
                             VRSTNDRD  VRSUNVER
                        VRSUNVB
                              VRGYCONF  VRGYRSL2
                     VRMGBIAS  VRMBIAS2
.
```

.

.

It should be noted that the one anomaly in the dendogram is the component TPobcver, which is an interface between the TP subsystem and the VR subsystem.

Based upon this study, Basili and Hutchens drew the conclusions that there is a mild correlation between size and control metrics and program changes, differences in individual performance have a large impact on the analysis, different development methods produce distinct models, statement count seems as good as any other metric, multiple regression models are good mostly for discounting team differences, data bindings and clustering provide a reasonable view of system modularity, and data hiding techniques influence the module hierarchy pattern.

Measurement across time

Measures are sometimes difficult to understand in the absolute. However, the relative changes in metrics over the evolution of the system can be very informative. This evolution may be within one development cycle of the product, e.g.,

requirements --> design --> code ,

or multiple versions of the product, e.g.,

$code_1$ --> $code_2$ --> $code_3$. . .

One approach for examining these metrics over different versions of the system is to keep track of the metrics, via a metric vector associated with each version. The vector characterizes the product at some point in time. It can be viewed it at various stages of development and maintenance to monitor how the product is changing. A set of bounds for those metrics can be provided to signal potential problems and anomalies. For example, a vector of metrics, M_1, M_2, . . .M_n dealing with various aspects of the product, i.e., effort, changes, errors, product dimensions, execution traits, environmental considerations, can be associated with each version of a system. Here product dimensions might include (decisions, a measure of the interaction of data across modules, a measure of the interaction of data within a module, size). Then if there is a dramatic change in one metric which is not consistent with the changes in another metric, a warning can be issued to the manager or developer.

As an example of tracking various metrics over time, we will examine a study by Basili and Turner. The approach used was an iterative enhancement life cycle model and the data available are the results of the analysis at the end of five of the seventeen iterations of the development.

Various metrics were used during various points in the development of a software product. These metrics and their values are shown in Figure 6.10. The product was a compiler for a structured programming language. It was about 6400 high level executable statements, 17,000 lines of source code including comments.

Iteration	1	2	3	4	5
Number of statements	3404	4217	5181	5847	6350
Number of procedures and functions	89	189	213	240	289
Number of separate compiled modules	4	4	7	15	37
Average number of statements per proc/func	38.2	22.3	24.3	24.4	22.0
Average nesting level	3.4	2.9	2.9	2.9	2.8
Average number of tokens per statement	5.7	6.3	6.6	7.2	7

Figure 6.10 STATISTICS FROM COMPILERS AT 5 SELECTED POINTS IN THE ITERATIVE PROCESS

Examining the changes of the values of various metrics across time provides insight into how the product is progressing. For example, after iteration 1, there was a major design change which accounted for a major change in the number of functions and procedures and their size. Over the five reported iterations, the size nearly doubled but the functionality more than doubled. This can bee seen from the fact that the number of operators and operands per statement increased with each iteration, possibly implying that the developers were able to understand the application better over time and write more powerful statements. What cannot be seen from the data is that the language in which the compiler was written (which was the same language as the one being developed) became more powerful over time, introducing string processing at level 4. As the developers changed the design, they learned more and more about modularity and so although the design remained fixed after the second iteration, the modularization changed as they learned more about the effects of adding new function to the system.

Dynamic Metrics

The product models and metrics discussed so far were static, i.e., the could be applied to the product without executing it. In this section we will discuss a few dynamic metrics, i.e., measurement that can be taken on the executing program. Clearly dynamic measurements depend upon the specific test cases run.

We will begin by briefly discussing test coverage metrics. There are a variety of such metrics based upon the aspect of the code being executed. The simplest of these metrics is **procedure coverage**, a measure of the number of procedures executed by the test suite. Measuring the coverage of a test suite, provides some insight into the test process and the set of tests. Coverage metrics can be used to evaluate completeness of testing, demonstrate where more testing is needed, allows us to compare

effectiveness of the test plan against the operational environment for the product, and can be used to evaluate the functional test plan.

Aside from procedure coverage, there are several classes of coverage. **Statement coverage** is a measure of the statements that were executed; **branch coverag**e is a measure of the number of branches that were executed both ways. Coverage is usually measured by a tool that instruments the compiler to calculate the particular coverage for any execution and then accumulates the results of several execution to provide a full coverage count.

To demonstrate the use of coverage metrics, Leon Stucki, used coverage metrics to provide insight into where more testing was needed. Figure 6.11 covers two examples, one in Pascal and one in FORTRAN. In the Pascal example, the coverage was accumulated of the results of 32 tests that were run on a product. Based upon these results, it was seen that only 23% of the input and output statements were covered. In an attempt to improve coverage of these statements, four new tests were generated whose specific purpose was to improve coverage in this category as well as in other categories. This approach supports the tester in evaluating the completeness of testing and demonstrating where more test cases were needed. In a second example, 68 tests were run and the resulting coverage for the different types of statements were used to provide the tester with an insight of how much testing was performed on the product.

PASCAL

# Test cases:	32	36
Subprogram coverage	.81	.92
Branch path coverage	.59	.67
I/O coverage	.23	.54
DO Loop entry	.92	.94
Assignment	.85	.91
Other executable	.74	.78
Code coverage	.70	.80

FORTRAN

#Test cases:	68
Subroutine coverage	.91
Function coverage	1.00
Branch path	.63
I/O	.35
DO-loop	.74
Assignment	.48
Other executable	.66

Figure 6.11 USE OF TEST METRICS

Another example of a study using coverage metrics for comparison and evaluation was performed by Basili and Ramsey. Their goal was to evaluate the functional test plan used in the SEL. Stated as a GQM goal:

GQM Goal:

Analyze the **acceptance test plan** in order to **evaluate and improve** it for future releases with respect to its **ability of the acceptance test suite to cover the operational use of the system** from the point of view of the **test developer**.

The environment was the NASA SEL and the project was a subset of a large satellite system. The experimental design was a single project/case study.

Process Questions

Process conformance:

What is the test methodology?
It is the standard methodology used in the SEL, supported by a test plan process.

How well is it being applied?
Since it is standard and doccumneted, most testers have used the methodology before and can apply it fairly well.

Domain conformance:

How well do the testers understand the application?
The application is reasonably well understood but there was some aspects of the system that were new.

Product Questions

Product dimensions:

What is the size of the system?
The subsystem used in this study consisted of 68 Fortran subroutines, 10,000 lines of source code, and 4,300 executable statements.

What is the size of the test suite?
There were 10 multi-part acceptance tests. It was not a rigorous sampling of the input domain but not trivial as each sub-test was quite involved.

What are the number of operational uses?

The first 60 applications of the system were captured.

Changes/defects:

How many faults were found during acceptance test?

How many faults were found during operational use? (8)

Context:

How was the system being used during operation?
The use was normal use by the scientists for whom the application was built.

Product Questions

Quality perspective: Compare the structural coverage of the acceptance tests and operational use of the system.

What is the procedure coverage for the acceptance test suite by test and in total?

What is the statement coverage for the acceptance test suite by test and in total?

What is the % of unique code exercised by each test?

What is the procedure coverage for the operational use of the system?

What is the statement coverage for the operational use of the system?

What is the overlap of the acceptance test and operational use coverage?

Is there anything different about the statements executed in operational test but not covered during acceptance test?
Product Questions

Feedback:

Is there any indication, based upon the coverage representation, to indicate whether reliability models can be applied during acceptance test to predict operational reliability?

Figure 6.12 contains the answers to questions about the structural coverage of the acceptance test plan. You should notice that only 75% of the procedures and 56% of the executable statements were covered by the test plan. Thus, 25% of the procedures and 44% of executable statements were not exercised during acceptance test. They may have been executed in system or unit testing. Also anyone

of tests t1, t2, t3, t6, t7, t9, or t10 could have been omitted since they added no new coverage. Note that that does not mean that test no new function. It should also be noted that 42.6% of the procedures and 18.1% of the code was executed by every test.

Case	**% Procedures Executed**	**%Executable Statements**	**% Unique Code**
t1	50.0	27.5	0.0
t2	50.0	27.2	0.0
t3	48.5	24.4	0.0
t4	60.3	37.9	4.4
t5	69.1	47.1	1.7
t6	67.6	42.7	0.0
t7	66.2	39.0	0.0
t8	66.2	45.6	1.0
t9	66.2	41.0	0.0
t10	66.2	40.2	0.0
Cumulative	75.0	56.0	
Intersection	42.6	18.1	

Figure 6.12 STRUCTURAL COVERAGE OF ACCEPTANCE TEST

Figure 6.13 provides the coverage data for the first 60 uses of the system. Note that 80.9% of the procedures and 64.9% of the executable statements were executed by the users of the system. Note that 27.9% of the procedures and 10.3% of the executable statements were executed by all of the operational cases.

	Procedures Executed (%)	Executed Statements (%)
Cumulative	80.0	64.9
Intersection	27.9	10.3

Figure 6.13 STRUCTURAL COVERAGE OF OPERATIONAL USE

To answer the questions of the overlap of coverage between acceptance test and operation, figure 6.14 provides the intersection of the two coverages. This helps us understand whether the acceptance tests are representative of the operational use. Clearly this must be true if acceptance test failures are used to predict operational failures. As can be seen from the figure the overlap was 55.7% and 8.4% percent of the statements covered by operational use was not tested during acceptance test. This lack of coverage of the acceptance test suite of operational use caused the following two questions to be asked: Was there anything different in the 8.4% of the code not covered during acceptance test, and were there any faults found during operational use to that code?

The mix of statements in the 8.4% and 55.7% differ in that it is twice as likely to execute a call or if in the 8.4%, otherwise one canÆt distinguish by structural coverage numbers. This might imply that the tested code was more complicated than the tested code. However, there were no faults were revealed in the 8.4%. It is possible that even though this code was not tested during acceptance test, it was test during unit or system test.

Coverage Overlap:

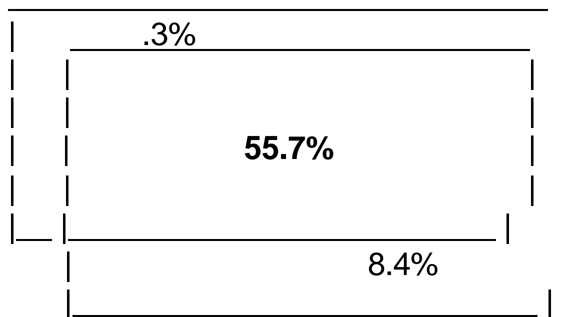


Figure 6.13 Overlap between Acceptance test coverage and operational use coverage

The authors drew the following conclusions from the study:

About 56% of code exercised by acceptance tests; 65% by operational use and there was a large overlap between both coverages. Thus the functional test plan reasonably effective, but could be refined for future releases. One obvious way to improve the plan would be to analyze the operational uses and generate new tests based upon them.

Acceptance test was reasonably representative of operational tests. Although the acceptance tests tended to cover a larger subset of the code for each run, the pattern was similar to operational use.

Since no faults were found in unexercised code, it appears that the code was tested prior to acceptance and that the overall test process was reasonably effective.

If acceptance tests randomized, reliability models may be used to predict operational reliability with moderate chance of success. The major drawback here is that there is probably insufficient data, in terms of failures found during acceptance test, to seed such a model.

Reliability Modeling

Reliability is a measure of how well a program functions relative to its operational requirements. It is usually viewed from a customerÆs perspective metric.

Reliability measurement can be use during system engineering to determine the best tradeoff between reliability and cost, schedule, etc., to optimize life cycle cost, and to specify reliability to the designer. It can be used for project management for progress monitoring, scheduling, and the investigation of alternatives. I provides a mechanism to support operational software management and the evaluation of software engineering management.

There are a variety of software reliability models. These include time dependent approaches, such as measuring the time between failures and failure counts in specified intervals (failure intensity) as well as time-independent approaches such as error seeding and input domain analysis approaches.

Some of the problems with the use of reliability models include the lack of clear understanding of their inherent strengths and weaknesses, the underlying assumptions and outputs, which are not fully understood by the user, and the fact that not all models are applicable to all testing environments.

Time-dependent models are based upon the belief that the number of failures in a specified time is random. This can also be stated as the time interval between failures is random.

Failure intensity is defined as the number of failure per unit of execution time, e.g., .05 failures per CPU hour. Mean time between failure is measured as the probability of failure free operation of a computer program for a specified time, e.g., the reliability is .9 for 24 CPU hours. Failure density depends upon the probability of fault introduction, the rate and time of fault removal, and the environment. Fault intensity and time between failures is assumed to be inversely proportional.

Time dependent reliability is usually defined as the probability of ôsatisfactoryö operation for a specified time in a specified environment. The concept of reliability modeling comes from hardware reliability, where the mean time to failure of items is measured to predict their life time, e.g., the mean time to failure for a light bulb. This concept has been mapped onto software.

One of the most commonly used time-dependent models in the literature is due to John Musa. The most basic form of the model is given below. It is based upon the assumptions that:

1. errors are distributed randomly through the program,
2. testing is done with repeated random selection from the entire range of input data,
3. the error discovery rate is proportional to the number of errors in the program,
4. all failures are traced to the errors causing them and corrected before testing resumes, and
5. no new errors are introduced during debugging.

The basic model is given by the formula

$$T = \frac{1}{KE} e^{Kt}$$

where E is the total errors in the system
t is the accumulated run time (starts @ 0)
T is the mean time to failure