

CHARACTERIZING DEFECTS

CHANGE AND DEFECT DISTRIBUTIONS

Software development environments vary with respect to a variety of environmental characteristics, e.g., application, organization, life cycle models, methods, techniques, experience, etc. Defect and change distributions can provide us with a signature of the project and the organization relative to those characteristics. That is, we can find patterns that can provide descriptive models that result from certain sets of characteristics and use those patterns to predict expected outcomes for projects within that class or modify project characteristics within our control to change those outcomes. For example, knowing the defect/change patterns of a class of projects can provide input into the adoption of various methods and tools for software development, evaluation and refinement.

Using these change and defect patterns can provide us with very useful information. We can understand the effectiveness of the various phases/documents in the life cycle, e.g., which phase was the source of the most errors, e.g., requirements, design, etc. or more specific issues, such as how expensive the requirements document was to update.

We can analyze the effect of the methods and tools used. Did a method which was supposed to minimize a certain class of error have that effect? Did it minimize the expensive errors of that class?

It can provide an understanding of the kinds of errors and faults we are making. What is the distribution between faults of omission and commission? What percent of the faults require a fix to more than one module? What percent of faults require the knowledge of more than one module to design the fix? What kinds of errors/changes are the most expensive? Are they errors/changes in the requirement, dealing with a certain functional capability? Are they errors we can do something about?

It can provide insight into the kinds of techniques that work best in uncovering faults and failures or classes of faults and failures. Are they faults that could be caught by an automated code analyzer, e.g., initialization of data? Are they failures that could be found by structural testing, e.g., boundary conditions on loops?

In this chapter we will examine a set of studies that have looked at various errors, faults and failures in different environments.

Endres IBM DOS/VS Study

One of the earliest studies on defects was performed by Endres [Endres 1975]. He studied a new release of the DOS/VS operating system (release 28). The update to the system consisted of the addition or modification of approximately 250K machine instructions and comments, composing about 500 modules. The average size of a module was 360 lines without comments, 480 with comments.

The study involves the analysis of 432 defects that were discovered during the internal test of the system. It should be noted that this study was conducted well before the IEEE standard definitions of error, fault and failure. The defects enumerated in this study were derived from problem reports generated during testing of the system, caused by failures of some kind from the system's operational behavior. In the terminology used in this book, they might be thought of as failures from system test. Unit and integration test are not included.

Goals of the study included an analysis of whether or not the interface between modules is a major source of failures and the benefits derivable from using a high level programming language versus assembly code. The distributions calculated include the number of modules affected by a defect, the number of defects per module, and a classification of the error domain of the defects.

Figure 5.1 offers a view of the number of modules affected by a failure. This data shows that most failures (85%) affected a change to only one module, i.e. only 15% of the failures required changes in more than one module. Using the definition that an interface fault is one which requires the modification of more than one module, this implies that the vast majority of faults are not interface faults. This is the same definition of interface fault that was used in the Weiss/Basili study reported in the previous chapter. In a latter study we will introduce another definition of interface fault.

Number of Failures	Number of Modules Affected
371 (85%)	1
50	2
6	3
3	4
1	5
1	8

Figure 5.1 Number of Modules Affected by a Failure

Figure 5.2 provides a view of the number of failures per module. Thus some modules are more fault ridden than others. In analyzing the most fault ridden modules, Endres found that they were usually larger than the norm, so that the fault rate was not out of line, or they were driver modules and thus more complex than the other modules from the point of view of call structure.

Number of Modules	Number of Failures/Module
112	1
36	2
15	3
11	4
8	5
2	6
4	7
5	8
3	9

2	10
1	14
1	15
1	19
1	28

Figure 5.2 Number of Failures per Module

In analyzing the domain of the error that created the fault, Endres divided the error cause in to three main categories, problem specific errors, implementation specific errors, and textual errors. Figures 5.3, 5.4 and 5.5 provide the categorized data. The purpose of this categorization was to

Cause	Percent of Errors
A1 Machine Configuration and Architecture	10
A2 Dynamic Behavior and Communication Between Processes	17
A3 Functions Offered	12
A4 Output Listings and Formats	3
A5 Diagnostics	3
A6 Performance	1
Total Percent of Type A Errors	<hr/> 46%

Figure 5.3 Problem Specific Errors (Type A)

Cause	Percent of Errors
B1 Initialization (of Fields and Areas)	8
B2 Addressability (in the sense of the assembler)	7
B3 Reference to Names	7
B4 Counting and Calculating	8
B5 Masks and Comparisons	2
B6 Estimation of Range Limits (for addresses and parameters)	1
B7 Placing of Instructions within a module, bad fixes	5
Total Percent of Type B Errors	<hr/> 38%

Figure 5.4 Implementation Specific Errors (Type B)

Cause	Percent of Errors
C1 Spelling Errors in Messages and	

Commentaries	4
C2 Missing Commentaries or Flowcharts (Standards)	5
C3 Incompatible Status of Macros or Modules (Integration errors)	5
C4 Not Classifiable	2
	<hr/> 16%

Figure 5.5 Textual Errors (Type C)

distinguish between failures that could have been prevented by the use of a high level programming language and those that should not have been affected by the use of such a language.

The conclusions that can be drawn from this data are that a high level programming language might have prevented 38% of the failures committed during development.

Weiss ARF Study

In this study [Weiss 1979], Weiss analyzes the effect on errors from using a methodology that involved the use of information hiding in defining modules. His analysis goals are an attempt to recognize the effect the design principles had on the classes and cost of errors. He also evaluates the effect of the use of a preprocessor for finding errors, as opposed to program execution or reading, and the particular source of the reading activity for those faults found during reading.

The project under study was a facility for simulating different computer architectures under development at the Naval Research Laboratories. The project size was 21K lines of FORTRAN source code with comments, consisting of 8 very large modules and 253 FORTRAN subroutines. The resources expended consisted of 9 people, several only part time, over 10 calendar months comprising 48 staff months. There were 143 defects reported from the end of design to one year after the end of development.

Designers and programmers reported defects and categorized them after correction. There was a validation of the defect classification by an error analyst and a calculation of the distributions and parameters of interest at various stages of project development.

A misunderstanding is defined as an error resulting from an incorrect assumption. For example, if the coder assumed that the user input routines removed delimiters from character strings containing user commands, resulting in a valid user command being rejected, then this would be an error caused by a misunderstanding of the design. Figure 4.6 gives the number and percent of misunderstandings due to various causes. The implication is that few errors were due to a misunderstanding of the interface design. More specifically, only 8 (6%) of the errors required understanding of more than one module, while 135 (94%) of the errors required understanding only one module. When comparing this with the Endres study, it must be remembered that the

Category	# of Errors	% of Total Errors
Requirements	8	6
Design		
Excluding Interfaces	27	19
Interface	9	6
Coding Specifications	18	13
Language	12	8
Coding Standards	3	2
Careless Omission	14	10
Clerical	52	36

Figure 4.6 Misunderstandings as a Source of Errors

definition of module is different, here there were only 8 information hiding modules, i.e. they were more like subsystems. However, since this was one of the major goals of the design method, to reduce the number of errors across modules, one would have to consider the method effective in achieving its goal.

It should also be noted that 46% of the errors (46 errors) were not due to misunderstandings at all but due to careless omissions or clerical mistakes.

In order to estimate the cost of the various errors, Weiss broke down the effort into three categories: easy: less than a few hours, medium: a few hours to a few days. and hard: more than a few days.

Category	# of Errors	% of Total Errors
Easy	120	84
Medium	22	15
Hard	1	1

Figure 4.7 Difficulty of Finding the cause of and correcting errors

Figure 4.8 classifies the category of misunderstanding with regard to the cost. From this figure we can see that the interface design errors were reasonable inexpensive. It might also be noted that even though there were a small number of errors due to a misunderstanding of the requirements, these errors tend to be expensive.

Category of Misunderstanding	Easy		Medium		Hard	
	#	%	#	%	#	%
Requirements	4	3%	3	2%	1	1%
Design						
Excluding Interface	19	13%	8	6%	0	
Interface	8	6%	1	1%	0	
Coding Specifications	14	10%	4	3%	0	

Language	9	6%	3	2%	0
Coding Standards	3	2%	0	0%	0
Careless Omission	13	9%	1	1%	0
Clerical	50	35%	2	1%	0

Figure 4.8 Difficulty of Error Isolation/Correction

Figure 4.9 gives the number of errors detected by each of the three different approaches to finding errors, program execution, reading of the coding specifications, and detected automatically via the preprocessor or compiler.

Category	# of Errors	% of Total Errors
Program Execution	58	40
Reading Code or Specifications	41	29
Preprocessor, Compiler	44	31

Figure 4.9 Methods of Error Detection

It should be noted that of the 58 errors detected by program execution 21 were hard or medium with regard to effort. With regard to reading all but one was easy. This could imply that early error detection, by reading makes error detection cheaper. It should also be noted that the figure of 29% of the errors found reading is a lower bound since there may have been unreported errors found from reading that occurred before the data collection began.

Weiss/Basili SEL Study

In the previous chapter we discussed the results of part of this study, the error domain patterns from a class of common projects. Here we will deal with the goal of categorizing changes [Weiss/Basili 1985]. The goal here was to characterize the changes made to projects over the course of development. More specifically, the GQM goal is

Analyze the projects in order to characterize the changes by class of change from the point of view of the manager.

There are several models of changes in the form of classifications. First we are interested in whether the change was a modification or an error. Error was subclassified into clerical and non-clerical because of the concern over the validity of the non-clerical errors reported. Another category is the source of modification, i.e., was the modification externally generated, e.g., from a requirements change or a change in the environment, or was it internally generated from a change in the design, the addition or deletion of debug code, or a change that was anticipated by the developers such as the addition of function at a later time because of an incremental development approach. There is also a need to understand the cause of the design change, i.e. is it to offer a clearer, simpler design, is it to provide the

user with better services in the form of a better interface, etc., is it for the purpose of optimizing the design. As with error data, the effort to change can also be categorized relative to categories of anticipated effort.

These models can be summarized as follows:

type of change: modification, non-clerical defect, clerical defect

source of modification: requirements, design, debug code, environment, planned enhancement, other

type of design modification: clarity, user services, optimization, other

effort to change: less than or equal to 1 hour, 1 hour to 1 day, 1 day to 3 days, greater than 3 days

Figure 4.10 shows the percent of changes by type of change. What can be seen from this table is that the number of modifications is at the same level as the number of non-clerical errors.

Type of Change	SEL1	SEL2	SEL3
Mods	36%	48%	54%
Non-clerical Errors	47%	44%	37%
Clerical Errors	17%	8%	9%

Figure 4.10 Percent of All Changes by Type of Change

Figure 4.11 shows the percent of changes by source of modification. From this table it should be noted that during the period that the data was collected, after the code was baselined, most of the changes are internally generated. In fact most of them have to do with changes to the design.

Source of Modification	SEL1	SEL2	SEL3
Requirements	3	29	26
Design	65	49	35
Debug	10	4	10
Environment	2	4	1
Planned Enhancement	20	12	27
Other	0	2	1

Figure 4.11 Percent of Changes by Source of Modification

Figure 4.12 shows the breakdown of the design modifications. Although these were generated by the developers, several of these changes may have been due to incomplete or missing requirements. For

example, design modifications of the type user services may be due to requirements that the developer know the user wanted but were not listed in the requirements document. It is possible that several of the optimization modifications could have lead to faults or failures of performance but there is no way of knowing at this point in time. Several modifications of clarity may have also lead to faults but were prevented before they could occur.

Source of Design Modification	SEL1	SEL2	SEL3
Clarity	8	11	17
User Service	21	10	12
Optimization	28	18	6
Other	3	1	
Unknown	5	9	

Figure 4.12 Percent of Changes by Source of Design Modification

This argues that the enumeration of defects as well as their categorization is highly dependent on the local environment. For example, the classes chosen in this study, as well as in the last two studies, were based not only on the local goals but on the local processes, environmental characteristics and when the data collection process begins and ends. What might be a defect in one environment is a modification in another and vice versa. Therefore it is very difficult to compare data from different environments.

Another problem that compounds the problems with comparison is the validity of the data. During this study, we analyzed the validity [Basili/Weiss 1985]. We analyzed how often a change report form needed to be corrected by the error analyst as a result of the data validation process. Figure 4.13 shows the number of forms that were corrected on each project as a percent of the original forms and the number of new forms generated based upon the error analyst recognizing that there were several different errors combined on one form. Since the three projects were begun in the order given, there is some sign that the developers became more accurate over time.

	SEL1	SEL2	SEL3
Percent of Forms Corrected	55%	51%	34%
Percent of Forms Generated	17%	36%	4%

Figure 5.13 Forms Corrected as Percent of Original Forms

Note that this data does not even take into account the number of missing forms, i.e. changes made for which forms were not filled out.

As an example of the kinds of mistakes made in filling out the forms, Figure 4.14 gives a sample of the kinds of changes made by the error analyst. Note that developers often miss-categorized non-clerical errors as clerical errors or modification but rarely the reverse, i.e., developers were more forgiving than the error analyst.

Initial Classification	After Validation	Number of Categorization Changes
Clerical error	Non-clerical error	46 (35% of non-clerical)
Modification	Non-clerical error	31 (23% of non-clerical)
Non-clerical	Clerical error	0
Change	Clerical error	0

Figure 5.14 Sample of Categorization Changes Resulting from Validation

Basili/Perricone SEL Study

This next study [Basili/Perricone 1984] examines the defects in a satellite planning study of software, reviews the results in the light of those from other studies, and analyzes the relationship between defects and complexity.

GQM goal:

Analyze the **life cycle process for a particular project** in order to
characterize it with respect to **errors and faults**
evaluate the results with respect to **those from other studies**
characterize the **relationship between errors and complexity**
from the point of view of the **experience factory**.

The environment is the NASA/GSFC, SEL. The application is different from the normal ground support software for satellites, it is a simulator for satellite planning studies which uses many of the same algorithms but applies them in a different context.

The system is 90K source lines of Fortran code consisting of 517 code segments, 370 Fortran subroutines, 36 assembly segments and 111 common modules, block data, and utility routines. The part of the system discussed here only deals with the 370 FORTRAN subroutines. 28% of the modules were new, developed specifically for this system and 72% were modified, adopted from a previous system.

The life cycle covered by the data is design through three years of maintenance and the requirements for the system kept growing and changing over the life cycle. The project used the same programmer/manager throughout the life cycle.

Defect data was collected starting with the base-lining of the code through the three years of maintenance. There are two definitions of defects used: faults and errors. Based upon the change report forms, the 215 faults were analyzed as 174 different errors. 49% of the faults were in modified modules and 51% of the faults were in new modules. Of the total changes to the system 38% of changes were modifications and 61% of changes were defect corrections.

The experimental design is a single project/case study but some comparisons can be done with prior projects from the environment (multi-project analysis), even though the application and time frames are different.

GQM Questions:

Process conformance:

What is the life cycle model?

The life cycle model is basically a waterfall model followed by an incremental development model in that the system requirements changed with use. Since many of the algorithms are standard, a fair number of previously defined code was reused which made the front end of the life cycle look very much like the standard SEL development. What was different was the amount of change made over time based upon the learning of the users with the needs of the simulation.

How well is it being applied?

The methods are quite standard and the basic process model is well understood by the developers.

Domain conformance:

How well do the developers understand the application?

Most of the developers were trained as mathematicians and had a reasonable familiarity with the application domain. Viewing the system as a simulator for planning studies rather than as a standard ground support system did provide a new perspective on the application domain.

How well do the developers understand the requirements?

The requirements document was not well developed and the system definition evolved over time. However, the basic algorithms were well understood even though the overall perspective of the system was new.

Quality perspective:

In what follows we will answer the following questions:

- What is the distribution of errors by error origin?
- What is the distribution of the number of modules affected by an error?
- What is the distribution of errors/faults per module?
- What is the distribution of errors/faults in new and modified modules?
- What is the distribution of faults by abstract fault classes?
- What is the distribution of faults by omission/commission?
- What is the distribution of faults by software aspect?
- What is the relationship between module size/complexity and fault rate?

Models used for the various defect distributions include:

Error Categories:

Change effect:

Number of modules changed to fix an error

Error Origin:

Requirements incorrect or misinterpreted

Functional specification incorrect or misinterpreted

Design error involving several components

- Design error in a single component
- Misunderstanding of external environment
- Misunderstanding of the programming language or compiler

Fault Categories:

Effort to Isolate/Fix:

- 1 hour or less
- 1 hour to 1 day
- 1 day to 3 days
- more than 3 days

Software aspect:

- Initialization - failure to initialize data on entry/exit
- Control structure - incorrect path taken
- Interface - associated with structures outside modules environment
- Data - Incorrect use of a data structure
- Computation - erroneous evaluation of a variable's value

Omission/commission:

- Omission - neglecting to include some entity in a module
- Commission - incorrect executable statement

Fault Density:

- Number of faults per KLOC

Figure 5.15 provides an overview of the number of distribution of module (FORTRAN subroutine) sizes. The table is set up to display the number of source lines of code and executable statements falling into various size increment of 50. This is done both for all modules and for modules with faults. You should notice that most of the modules are relatively small.

Number of Lines	All Modules		Modules with Faults	
	Source	Executable	Source	Executable
0-50	53	258	3	49
51-100	107	70	16	25
101-150	80	26	20	13
151-200	56	13	19	7
201-250	34	1	12	1
251-300	14	1	9	0
301-350	7	1	4	1
351-400	9	0	7	0
> 400	10	0	6	0
Total	370	370	96	96

Figure 5.15 Distribution of Module (FORTRAN subroutine sizes)

Figure 5.16 shows the number of modules affected by an error. To some extent this duplicates the analysis done by Endres with similar results. He showed that 85% of the changes to modules due to errors affected only one module. This study shows that 89% of the changes due to errors affect only one module supporting Endres's hypothesis that few errors affect more than one module. Errors (174) not faults are used in this analysis. It should be noted that there are differences in the data used in this and the Endres study. First, the definitions of errors is slightly different. Second the time frame for which the error data is collected is different (system test versus system test through maintenance). However, these probably have little effect on the common interpretation of the results.

# ERRORS	# MODULES AFFECTED
155 (89%)	1
9	2
3	3
6	4
1	5

Figure 5.16 The number of modules affected by an error

Figure 5.17 shows the number of faults per module. As with the Endres study some modules are more error prone than others. (Note that the same cautions about the inconsistency of the data from the two studies still holds here, with similar beliefs about the commonality of the results.) The figure also breaks the modules down into new and modified modules. There are 49 new modules with faults and 47 modified modules with faults. Since there are almost three times as many modified modules than new modules, we see that the new modules were more error prone than the modified modules. The number of faults used to calculate the data from this table is the 215 faults. The asterisks point out the most fault prone modules.

# Modules	New	Modified	#Faults/Module
36	17	19	1
26	13	13	2
16	10	6	3
13	7	6	4
4	1	3	5
1	1		7

Figure 5.17 The number of faults per module

Figure 5.18 shows the percent of errors by error origin for this project broken down by new and modified modules. This distribution shows that the majority of errors (36%) were associated with the functional specification, and within this category, the majority were associated with modified modules (24%). This might be due to the fact that the reused modules were taken from a system with a different application and even though the basic algorithms were the same, the functional specifications for those algorithms were wither not well-enough or appropriately defined to be used under slightly different circumstances. If we consider EndresÆ type A errors (problem specific) to be predominantly errors involving the requirements and functional specification, this data might also be used to support EndresÆ result.

Error Origin	New	Modified	Total	
Requirements		12.5%	4%	16%
Functional Specification		12%	24%	36%
Design/Code of Multiple Components		6%	1%	7%
Design/Code of a Single Component		12.5%	10%	22%
Language		0%	0%	0%
Environment		0%	0%	0%
Clerical error		6%	6%	12%
Error Due to a miss-correction of an error	4%	2%	6%	
Other		0%	0%	0%

Figure 5.18 Percent errors by Error Origin

Figure 5.18 shows the percent of nonclerical errors by error origin for this project (labeled SEL4) and compares it with the project SEL 2 given from the Weiss/Basili study. The appropriate subset of the data have been modified to make it consistent with the other SEL data error origin categories in Figure 5.18 in order to perform the comparison. One can see the from examining the figure that the error origin distribution is dramatically different. The major reason for the difference is that for the ground support systems, the application, and the requirements are well understood and the developers had a reasonable amount of experience with the application. Thus there are fewer errors in the requirements and functional specification. In this project, the requirements and the application are less well understood making this a new class of project and leading to a large percent of errors in the front end of the life cycle. Also the programmers on the standard project are less experienced than the programmer here, leaving more errors in the design and coding of a single component.

Error Origin	SEL4	SEL2
Requirements	19%	5%
Functional Specification	44%	3%
Design/Code of Multiple Components	7.5%	10%

Design/Code of a Single Component	22.5%	72%
Language	0%	8%
Environment	0%	1%
Other	0%	1%

Figure 5.19 Comparison of Error Origin with other projects in the SEL

Figure 5.20 gives the effort distribution for the cost of isolation and fix of the faults for both new and modified modules. It appears that it is more expensive to isolate and fix faults in modified modules than new modules, i.e. 27% of the modified modules took more than 1 day to fix, while 18% of the new modules took more than a day to fix. Thus although the existence of available modules can shrink the cost of coding, the amount of effort needed to correct faults may use up some of that savings. How much clearly depends upon the quality of the old code.

Cost of Isolate/Fix	New	Modified	Total	
1 hour or less	21%	15%	36%	
1 hour to 1 day		11%	8%	19%
1 day to 3 days		3%	15%	18%
more than 3 days		15%	12%	27%
Total		50%	50%	

Figure 5.20 Cost to Isolate and Fix New and Modified Modules

It should be noted that the definition of interface faults given here is different from the definition for interface faults in the Endres and earlier SEL study. Here an interface fault (which we will call a **design interface fault**) is a problem associated with structures outside a module's environment. In the prior studies, an interface fault was defined as a fault in which more than one module needed to be changed (which we will call an **implementation interface fault**). It is clear that a design interface fault does not involve a change to more than one module. For example, if the calling parameters of a function do not match the formal parameters of a called procedure, it is a design interface fault since one is required to know about both units, but it may not be an implementation interface fault since one may only have to change the parameters in one unit.

Figure 5.21 gives the percent of faults by software aspect and omission/commission for both new and modified modules and in total. From a brief look at the distributions, it is clear that the largest percent of faults involves interface (39%), control flow is more of a problem in new modules (28% vs. 8%), data and initialization are more of a problem in modified modules (38% vs. 18%).

This data might imply that the basic algorithms for the modified modules were correct but needed some adjustment with respect to data values and initialization for the application of the old algorithm to the new application.

The fact that there are a small number of omission faults in modified modules might imply that the modified modules were more complete since any omission faults for the general algorithm were already detected in the prior application.

	Commission		Omission		
	New	Modified	New	Modified	
Initialization	2	9	5	9	
Control	12	2	16	6	
Interface	23	31	27	6	
Data	10	17	1	3	
Computation	16	21	3	3	
	28%	36%	23%	12%	
	64%		35%		
	New	Modified	Total		
Initialization	7	18	25 (11%)		
Control	28	8	36 (16%)		
Interface	50	37	87 (39%)		
Data	11	20	31 (14%)		
Computation	19	24	43 (19%)		

	115	107	222		

Figure 5.21 Classification of faults by software aspect and omission/commission

Figure 5.22 gives the fault rate for modules in different size classes. It includes all module. It is clear from the data that the fault rate decreases as size increases. This appears to be a strange phenomenon, counter to the theory that smaller is better. However, if one thinks about it for a little while, the results are not so strange. If one takes æsmaller is betterÆ to its logical conclusion, it would imply one line modules are best. But clearly, this is not true.

There are many possible explanations for this phenomenon. The majority of modules examined were small biasing the result, or larger modules were coded with more care because the programmer was more worried about their correctness, or the faults in smaller modules were more apparent and were more susceptible to be caught by testing. Since the fault data did not include much unit test data this latter explanation seems less plausible than the others.

Module Size	Faults /1000 Lines
50	16.0
100	12.6
150	12.4
200	7.6
> 200	6.4

Figure 5.22 Faults /1000 executable lines

Another explanation, supported in part by the data, is that since interface faults are spread across all modules and interface faults dominate, smaller modules have a higher fault rate because of interface complexity. However, this would imply that as modules became very large in size, the internal complexity of the module might outweigh the interface complexity and the trend would reverse.

The real question is what size modules are best? We have a guide that says modules of one page size are easy to read because we don't have to flip the page. But that has nothing to do with the number of faults generated while developing a system. What are the size factors that might affect fault rate. Clearly, there are several, e.g., the natural size of an algorithm, the ability of the programmer, the language in which the algorithm is being expressed, etc. You will note that very few modules were over 400 line of code. It might be that a professional programmer naturally limits the size of the module when the internal complexity dominates the interface complexity. Thus, there should be a caution on setting size limits on modules.

You will note that one of the potential explanations was not that the complexity of the larger modules was lower than that of the smaller modules. This is because, based upon the data from Figures 5.24 and 5.25, the average cyclomatic complexity grew faster than size.

Module size	Average Cyclomatic Complexity
50	6.0
100	17.9
150	28.1
200	52.7
>200	60.0

Figure 5.24 Average cyclomatic complexity for all Modules

Modules Size	Average Cyclomatic Complexity	Faults /1000 Executable Lines
50	6.2	65.0
100	19.6	33.3

150	27.5	24.6
200	56.7	13.4
>200	77.5	9.7

Figure 5.25 Size, Average cyclomatic complexity, and fault rate for faulty modules

To summarize the results from this study it is clear that defect analysis provides useful information to the developer and the organization. From this study we saw that a new application with changing requirements change the standard defect profiles for the environment. Insights were provided into the different faults committed in new and modified modules.

Modules size is really an open question with respect to errors. Here we saw that the larger the module (within limits) the less fault prone the module. As a consultant to several companies, when such data has been available, I have asked the companies to perform the same analysis on their data. I have always found the results to be the same, i.e. fault rate goes down with size, but very few of the modules are unusually large. It is clear that we are still not ready to put artificial limits on module size without more analysis and understanding of the effect. However, it should be noted that this is only one consequence of module size. There may be other reasons for limiting the size of modules, e.g., ease of modification, readability ,etc. We clearly do not know what the effect of module size has on any of these parameters.

This study points out that we need to better understand the product, process, environment and the interaction of the three in order to establish guidance for developers and managers. More studies must be made, more data must be collected.

Basili/Weiss A7 Requirements Document Study

This next study [Basili/Weiss 1981] examines the effect of a methodology on the defects associated with the development of a requirements document.

GQM Goal:

Analyze the **requirements document** in order to **evaluate the affect of the methodology used to develop it** with respect to the **correctness** and **ease of modification** from the point of view of the **requirements writers** and the **experience factory**.

Environment: The document was developed at the Naval Research Laboratory. The project is an on-board flight program for the A-7 aircraft, a real-time, interactive system for the TC-2 computer (16K 16 bit words).

The data was collected after document was baselined. The experimental design is a single project/case study. This does not represent a complete analysis of the data but only the results of the first 15 months

after the first change was made to the document, after it was baselined. Thus this data should be viewed as indicative rather than complete.

Process Questions:

Process conformance:

What is the requirements development methodology?

The methodology used formal specifications in the form of a state machine model.

How well is it being applied?

The developers were experimenting with the methodology.

Domain conformance:

How well do the developers understand the application?

The developers had minimal expertise in the application domain.

How well did the developers understand the user needs?

The system was a redevelopment of an existing system and so had available the operational version of the previous system and had available the maintainers of the current version for questions.

Product Questions

Product dimensions:

What is the size of the requirements document?

It consisted of 462 pages.

Cost:

What is the staff effort expended in producing the document?

It took 17 staff months.

What is the staff effort expended in making the changes?

It was 11 staff weeks.

What is the total staff effort expended in development during the time the data was collected?

It was 122 staff weeks.

What is the calendar time for development during the time the data was collected?

It was 15 months.

Changes/defects:

How many changes are there to the document?

There were 88 changes in all.

How many of the changes are faults?

There were 79 faults, 18 of which were clerical.

What is the distribution of nonclerical faults in the requirements document by document fault type (i.e., ambiguity, omission, inconsistency, incorrect fact, wrong section)?

See Figure 5.26.

Document Fault Type	% of Nonclerical Faults
Ambiguity	5%
Omission	31%
Inconsistency	13%
Incorrect Fact	49%
Wrong Section	2%

Figure 5.26 Nonclerical Document Faults by Type

This figure shows that most of the faults are incorrect facts. 80% of the faults (omission and incorrect fact) could be detected by comparing the document with other sources. Relatively few faults (18%) are detectable by self-consistency checks (inconsistency) or by trying to find alternate meanings (ambiguity).

Context:

How is the document being used?

The document was used as a design reference by the designers and to a small degree by the maintainers of the original system.

How was the need for change discovered?

The need for change was discovered based upon a variety of uses of the document, including various reviews. See Figure 5.27 for the breakdown. The biggest use even at this part of time seems to be as a design reference. This implies that a consistent, complete, correct and unambiguous document is important and work maintaining.

Use of Document	% of Change
Review by Authors	23%
Review by Non-authors	10%
As Maintenance Reference	2%
As Design Reference	45%
As Coding Reference	1%

Other 19%

Figure 5.27 Discovery of Need for Change

Quality perspective: Ease of change

One of the goals set by the original requirements writers was that the document be easy to change. The model proposed for ease of change implies that the changes are inexpensive to make and contained in one section of the document.

What is the distribution of the types of changes?

Types of Changes	% of Changes
Original Error Detection	85%
Complete or Correct a Previous Change	6%
Reorganize	2%
Other	7%

Figure 5.28 Types of Changes

What is the distribution of changes by staff time to make the changes?

Effort	Range	% of Changes
Trivial	<= 1 hour	68%
Easy	1 hour to 1 day	26%
Moderate	1 day to 1 week	5%
Hard	1 week to 1 month	0%
Formidable	> 1 month	1%

Figure 5.29 Effort to Change

It can be seen that most of the effort to make changes and correct faults was in the trivial category. However, one formidable fault had been detected. This fault took six staff-weeks of effort to correct, far more than any other change. The total effort to make all changes was 442 staff-hours or about 11 staff-weeks. Note that without the formidable fault, the effort would be about 202 staff-hours or about 5 staff-weeks. The average effort to make a change was 5 staff-hours and the average to correct a fault of any type was slightly higher, 5.4 staff-hours. Without the formidable fault, these figures are sharply reduced, becoming 2.3 and 2.4 staff-hours respectively. The modes of the effort distribution for changes and faults are both .5 staff-hours.

What percent of the changes were confined to one section of the document?

See Figure 5.30

Sections of the Document	% of Changes
One Section	85%
More than one Section	15%

Figure 5.30 Confinement of Changes

Most of the changes (85%) were confined to one section of the document. It should be noted that the formidable fault was confined to a single section. Analysis of the effort for single section changes compared to multi-section changes shows that on the average the latter required about 27% more effort than the former, 6.1 vs. 4.8 staff hours. Without the formidable fault, about 310% more effort was required for multi-section changes over single section changes (6.1 vs. 1.7 staff- hours).

Without a basis for comparison, it is difficult to conclude with a high degree of confidence that the requirements document was easy to change. However, based upon figures 5.29 and 5.30, one might conclude that there were indications that the document was well structured and the effort to change, except for the formidable fault, was minimal and thus the document was easy to change.

Quality Perspective: The document is worth maintaining

The model to determine that the document is worth maintaining includes the cost of maintenance and the need for maintaining the document. Questions include:

What is the cost of maintaining the document?

Is the document used effectively?

As was discussed above, the cost of change was small, in fact most of the effort associated with changes needed to be expended whether the document was changed or not, since much of the effort went into understanding the changes that needed to be made. There were indications that the document was heavily used as a design reference. In fact, use of the document in this way uncovered 49% of the faults in the document. Thus one could conclude that the document was worth maintaining, a fact that many organizations do not set as a goal.

Based upon this study, the authors concluded that the data collection methodology, which was one of the first applications of the Goal/Question/Metric paradigm was successful on organizing the data and providing partial data back to the developer that was useful feedback to them. However, they noted that data analysis always seems to generate new questions of interest.

In their analysis of the A-7 requirements document, they concluded that: the document is relatively more consistent and precise than complete and correct, the document is well-structured, i.e., most changes

are confined to single sections, the document is easily maintained, i.e. there appears to be a small effort to make changes, and the document is worth maintaining, i.e. it was heavily used as a design reference.

EVALUATION OF FUNCTIONAL ACCEPTANCE TEST PLAN

Goal:

Purpose: Evaluate the acceptance test plan in order to improve it for future releases

Perspective: Examine the ability of the acceptance test suite to cover the operational use of the system from the point of view of the test developer

Environment: NASA SEL

Subset of a large satellite system

Experimental design: Single project/case study

Basili/Ramsey

Process Questions

Process conformance:

What is the test methodology?
(standard methodology used in the SEL)

How well is it being applied?
(testers have used the methodology before)

Domain conformance:

How well do the testers understand the application?
(reasonably well)

Product Questions**Product dimensions:**

What is the size of the system?
(68 Fortran subroutines, 10,000 lines of code
4,300 executable statements)

What is the size of the test suite?
(10 multi-part acceptance tests,
not a rigorous sampling of input domain but not trivial)

What are the number of operational uses? (60 uses)

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?
(normal use)

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?

STRUCTURAL COVERAGE OF ACCEPTANCE TEST

44% of executable statements were not exercised in acceptance test. They may have been executed in system unit testing.

STRUCTURAL COVERAGE OF OPERATIONAL USE

Structural Coverage of
60 Operational Usage Cases

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

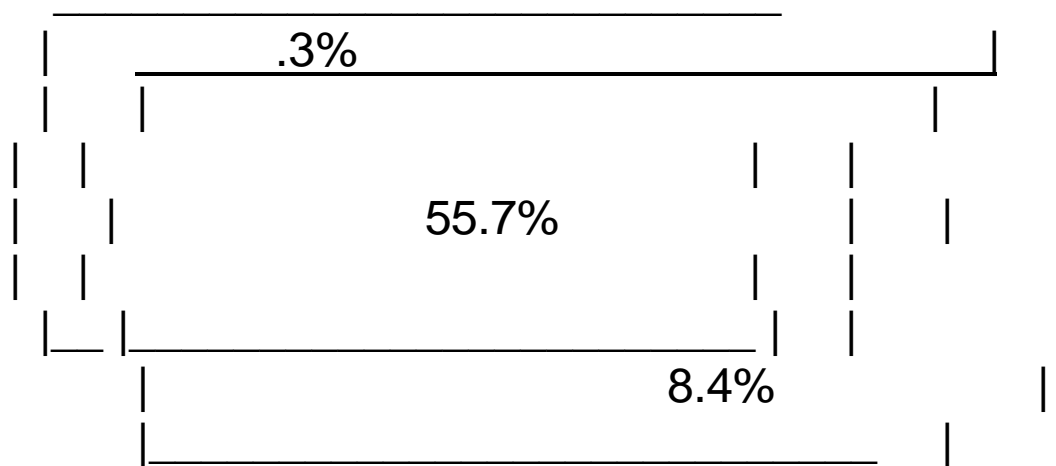
10% of the code was executed by all of the operational cases

ARE THE ACCEPTANCE TESTS REPRESENTATIVE OF OPERATIONAL USAGE?

Must be true if acceptance test failures are used to predict operational failures

Coverage:

Acceptance test



Operational test

Representation:

The mix of statements in the 8.4% and 55.7% differ
Twice as likely to execute a call or if in the 8.4%

Otherwise can't distinguish by structural coverage numbers

However, no faults were revealed in the 8.4%

OBSERVATIONS

Functional test plan reasonably effective, but could be refined for future releases.

About 56% of code exercised by acceptance tests; 65% by operational use.

Acceptance test reasonably representative of operational tests, no faults found in unexercised code.

If acceptance tests randomized, reliability models may be used to predict operational reliability with moderate chance of success.