A SURVEY OF SOFTWARE FAULT SURVEYS

Brian Marick[1]
Motorola, Inc.

$Revision: 1.4 $
$Date: 90/12/21 14:59:57 $

A number of people have published studies of faults found in software systems. This report summarizes the results of many of those studies.

Keywords: software faults, software errors, software bugs

[1] Brian Marick is the recipient of a Motorola Partnerships in Research Grant.

CHAPTER 1

Introduction

This introduction describes major trends in the studies. The trends were derived from the summaries given in the next chapter. An appendix briefly describes the studies surveyed.

I use the definitions given in [IEEE83]. An *error* is a mistake made by a person. An error manifests itself as one or more *faults* in some text, such as a requirements document or a program. Faults may be detected by direct observation (in, say, a code read), or they may be found by observing the *failures* they cause. One fault may cause several failures. (Or it may cause none.)

1. Results from the Literature

This section summarizes various studies of faults from the literature. Because there are no standard ways of categorizing or describing fault data, detailed comparisons are not possible. However, certain trends appear:

Faults in programming logic (path selection) are common.

Many studies [Dniestrowski78], [Potier82], [Lipow79], and [Motley77] find that faults in the programming logic (the decisions made by the program/design) are the single largest type of fault. [Rubey75] doesn't find it as high, but notes that those faults tend to be serious, as does [Glass81]. However, notice that this category is the *smallest* in [Basili87] -- perhaps because the application area was quite well understood.

Faults of omission are important.

In general, most faults were caused by doing the wrong thing (faults of commission); fewer were caused by failing to do something (faults of omission). However, [Glass81], who studied only faults found after delivery, says that omitted logic is the most likely fault to survive. He characterizes omitted logic as "code not as complex as required by the problem" -- that is, missing code (not just boolean operations).

The distribution of faults of commission and omission is not uniform. [Basili84a] finds that most interface faults were omissions. Data handling faults were overwhelmingly faults of commission, as were computation faults. [Ostrand84] found that most data definition faults were commission, data handling was balanced, and that 81% of faults associated with decisions were omitted code.

Data handling is more error-prone than computation.

Data handling faults are high in [Basili87] and [Glass81] (especially if you include initialization), and of middling frequency in [Rubey75], [Basili84a] (high if you include initialization), [Potier82], [Lipow79], [Motley77], and [Ostrand84]. They are of low frequency in no studies. Contrarily, computation faults (arithmetic, boolean expressions) are of low to middle frequency.

Static and dynamic detection are equally effective.

Most surveys found a rather even split between faults detected by dynamic testing (executing the program) and static analysis (typically code reads and walkthroughs). The conventional wisdom (that programmers are less effective than independent testers at testing their own code) is borne out.

Modified modules are no more error-prone than new modules.

All three studies that considered the matter reported that the fault density of new and changed modules was the same. [Basili84a] reports that the fault types differ, though.

There is evidence that both small and large modules are more error-prone than medium-sized modules.

Both [Basili84a] and [Shen85] note that smaller modules have higher fault rates. [Withrow90] confirmed this result, but also found that the fault rate then rose as modules became larger. The minimum rate was at about 251 lines / module for this sample of Ada programs.

No conclusions about development phases are possible.

The percentage of faults caused during coding ranges from a high of 83% to a low of 26%.

In abstract descriptions, beware of incorrectness, omissions, and inconsistencies, in that order.

In requirements / functional specifications, both [Bell76] and [Basili81] show that incorrect statements are most common, followed by omissions, followed (distantly) by inconsistencies. [Basili81] reports that inconsistencies took the longest to fix, followed by incorrect facts, followed by omissions.

Bug fixes cause a small number of new bugs.

The percentage of bugs caused by changes is low, around 10%.

CHAPTER 2

Details

2. Cross-Phase Summary Data

There were, in general, two sorts of categorizations: categorizations that span phases in the development process and categorizations within phases. This section describes the first.

[Dniestrowski78] gives the following faults for a "weak complexity" (largely arithmetic/boolean expressions) module with module with 3445 source instructions and a "great complexity" (real-time management, I/O drivers, etc.) module of 3475 source instructions.

	Weak complexity		Great complexity	
Fault	Totals	Percentages	Totals	Percentages
Arithmetic or Boolean Expression	3	4%	1	1%
Programming Logic	24	32%	56	50%
Software Interfaces	3	4%	8	7%
Data Declarations	19	25%	13	12%
Interrupt Handling	0	0%	7	6%
Hardware Interface	0	0%	11	10%
Documentation Standards	20	27%	9	8%
Realtime monitor interactions	1	1%	1	1%
I/O operations	1	1%	1	1%
Miscellaneous	4	5%	5	4%

Notes:

(1) Categories like "programming logic" are applied to designs and specifications in the original paper.

(2) One category is missing from the original table; I assume it is "miscellaneous".

In [Rubey75], it's not always clear what phase an error is made in, so I present all the data here.

Fault Category	Total	Serious	Moderate	Minor
Incomplete or erroneous specification	28%	11%	17%	43%
Intentional deviation from specification	12%	5%	13%	14%
Violation of programming standards	10%	1%	5%	17%
Erroneous data accessing	10%	21%	15%	2%
Erroneous decision logic or sequencing	12%	24%	17%	3%
Erroneous arithmetic computations	9%	13%	15%	3%
Invalid timing	4%	8%	5%	1%
Improper handling of interrupts	4%	8%	6%	0%
Wrong constants and data values	3%	8%	4%	1%
Inacurate [program] documentation	8%	0%	2%	16%
Total	100%	14%	40%	46%

[Weiss79] presents the following:

Fault Category	Number	Percent
Requirements	8	6%
Design (excluding interfaces)	27	19%
Interface	9	6%
Coding specifications [detailed design]	18	13%
Language	12	8%
Coding standards	3	2%
Careless omission	14	10%
Clerical	52	36%

NOTES:

(1) In all save clerical errors and careless omissions, the faults is a misunderstanding. For example, a "Language" fault might be a misunderstanding of a Fortran construct.

(2) Detailed design was done by writing in a high level language (e.g., Bliss or an Algol-like language) and translating into Fortran.

(3) A sample careless omission would be an omitted declaration.

[Basili84a] has faults divided up among two axes. The first axis is a set of fault categories:

(1) Initialization: failure to initialize or reinitialize a data structure properly upon a module's entry/exit.

(2) Control: faults that caused an incorrect path to be taken.

(3) Interface: incorrect use of a structure existing outside of the module's local environment.

(4) Data: incorrect use of a data value (wrong variable, wrong subscripts, etc.)

(5) Computation.

The second axis is faults of commission vs. faults of omission. A fault of commission is, for example, an incorrect executable statement. Forgetting to include some entity within a module would be a fault of omission.

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

Totals were 64% faults of commission, 35% omission. In [Basili87], the spread was 76% commission, 22% omission. (This is fewer omissions than average for his studies, perhaps because of a high level of code reuse.) In [Basili85a], the spread was 48% commission, 52% omission.

Data from [Ostrand84] can be reported in a similar way. For all the faults, 43% were incorrect (commisson), 54% of the faults were omissions, and 1% of the faults were due to superfluous code. This table shows the breakdowns for coding faults:

	Commission	Omission	Superfluous code
Data definition	68%	32%	0%
Data handling	50%	45%	5%
Decision	35%	65%	0%
Decision plus processing	3%	97%	0%

[Basili87] also produced this fault categorization:

Category	Percentage
Control	13%
Global Data Interface	13%
Other Interface	20%
Data	30%
Computation	16%

[Potier82] reports the following for their study and two others:

category	[Potier82]	[Lipow79]	[Motley77]
computational	6%	9%	9%
logic	38%	26%	26%
I/O	2%	14%	16%
data handling	15%	18%	18%
interface	19%	16%	17%
data definition	19%	3%	1%
data base	1%	7%	4%
others	0%	7%	9%

[Glass81] reports this data:

category	Project A	Project B	Total
Omitted logic (existing code too simple)	36	24	60
Failure to reset data	17	6	23
Regression fault	5	12	17
Documentation in error (software correct)	10	6	16
Requirements inadequate	10	1	11
Patch in error	0	11	11
Commentary in error	0	11	11
If statement too simple	9	2	11
Referenced wrong data variable	6	4	10
Data alignment error	4	3	7
Timing fault causes data loss	3	3	6
Failure to initialize data	4	1	5

(Note that some faults fall in more than one category.)

The most likely fault to survive is omitted logic -- code not as complex as required by the problem. (A typical sort of fault of this type would be "if (A)" when a correct test would be "if (A and B)".) Such errors are made in the detailed design or coding phase. [Ostrand84] reports that, for faults involving a decision, 81% were due to omitted code and 19% were due to incorrect code.

[Ostrand84] presents the following:

Fault Category	Number	Percent
Data definition -- code which defines data	56	32%
Data handling -- code which initializes or stores data	38	22%
Decision (alone) -- branching code	31	18%
Decision & Processing -- branching code plus executed body	32	18%
System -- program’s environment	12	7%
Documentation	2	1%
Unknown	2	1%

[Perry85] and [Perry87] concentrate on interface faults. Multi-file interface faults require changes to a header file or more than one file, whereas single-file interface faults require changes to only one file. 36% of faults were multi-file interface faults; 33% were single-file interface faults.

Fault Category	Multi-file	Single-file	Weighted Average
Data	18.8%	10.2%	14.7%
Functionality	18.8%	14.3%	16.7%
Error Handling	18.8%	22.5%	20.5%
Construction	12.9%	2.0%	7.7%
Inadequate Postprocessing	10.6%	10.2%	10.4%
Interface Misuse/Support	10.6%	24.5%	17.2%
Timing/Performance	4.7%	0.0%	2.5%
Coordination	4.7%	16.3%	10.3%

Data faults included data structures of inadequate size or with missing fields, initialization faults, and violation of data constraints. Functionality faults included assumptions that some unprovided function was provided, disagreements about location of function, and changes or additions because of changing needs. Error handling included both improper or missing error handling and also additions because of changes in other units. Construction faults are related to problems with the interface and implementation (typically associated with #include files). Inadequate postprocessing usually means failure to free working memory. Interface misuse/support includes interface misunderstanding, an interface that did not fully support its specified function, and hardware interfaces. Coordination faults occurred because not all modules requiring an update were changed.

3. Phase Data

Many of these studies predate [IEEE83], so each may mean something different by "requirements specification". My best effort was to relate the data to the following definitions, which are close to the IEEE definitions.

The *requirements specification* describes what the system *must* do. The *functional specification* describes what the system *does*. It is a complete definition of the observable behavior and interface of the system (what the customer can see). It is an extension of the requirements, in that it provides more details and perhaps more function. ([IEEE83] provides several variant kinds of specifications.)

The specification is refined in terms of successively more detailed designs. Conventionally, the first is called the *architectural design*. It provides the first level decomposition of the system by breaking it into components. (In [IEEE83] terms, it provides the *framework* for the system.) It should provide a complete functional specification for the components. The last of the designs is the *detailed design*, which is sufficiently complete to be implemented.

The following table shows the faults from the different studies, broken down into phases. Because of the difficulty in mapping phases, I've also lumped all pre-coding faults into the "ALL" category. The numbers don't add up to 100% for all studies because of "miscellaneous" categories and faults that were not related to phases.

	Req.	Func. Spec.	Arch.	Det. Des.	ALL	Coding
[Boehm75] avg				64	64	36
[Boehm75] max				73.7	73.7	26.3
[Boehm75] min				35.6	35.6	64.4
[Endres75]		46			46	38
[Rubey75]		40			40	60
[Schooman75]		4	10		14	71
[Dniestrowski78] WC		4		30	34	66
[Dniestrowski78] GC				30	30	70
[Herndon78]		19			19	58
[Potier82]	9.5	26.1	31.8		67.4	28.1
[Ostrand84]					14	79
[Perry85]					31.8	68.2
[Perry87]					16.3	83.7

Notes:

(1) For [Boehm75], I report the average for all 5 projects and also the two extremes. It's interesting to note is that the project with the greatest percentage of design faults was the largest, and the project with the fewest was the smallest. However, the trend doesn't hold for the three other projects.

(2) For [Rubey75], "specification" might mean anything prior to coding. Some of the coding faults might actually be design faults.

(3) For [Dniestrowski78], WC means a module with "weak complexity" (largely arithmetic and boolean expressions), whereas GC means "great complexity" (real-time management, I/O drivers, etc).

(4) For [Potier82], I've put "design specifications" under detailed design, though they may apply to architectural design. Their "requirements definition" likely describes a great deal of the functional interface, so those faults might also fall into the specification category.

[Basili84a] has to be reported separately because coding and detailed design faults are lumped together. (This is perhaps due to the development environment, in which there was not always a separate detailed design document -- see [Weiss85] -- but it may also be because, as [Glass81] points out, it's often difficult to classify faults into one or the other.) The same is true of the SEL1, SEL2, and SEL3 projects reported on in [Weiss85], except that coding faults may also appear in the architectural design (design of more than

one component).[1]

	Req.	Func. Spec.	Arch.	Det. Des.
[Basili84a] planning	19	44	7.5	22.5
[Basili84a] support	5	3	10	72
[Weiss85] SEL1	2	14	7	67
[Weiss85] SEL2	5	3	4	78
[Weiss85] SEL3	6	5	24	57

Notes:

(1) In [Basili84a], the large number of specification faults is taken to be "due to the fact that the reused modules were taken from another system with a different application. Thus, even though the basic algorithms were the same, the specification was not well-enough defined or appropriately defined for the modules to be used under slightly different circumstances". The other, better understood, support application shows a different distribution.

(2) In [Weiss85]/[Basili87], the large number of single-component faults may be caused by the inexperience of the personnel.

3.1. Requirements/Specification/Architectural Design Faults

[Basili81] reports on faults made in requirements specification, a document which describes external interfaces, data items, and functions. The fault categories were as follows:

Fault categories (excluding clerical)			
	Total	External Interfaces	Functions
		(Data Items)	
Ambiguity	5%	7%	4%
Omission	31%	41%	21%
Inconsistency	13%	15%	7%
Incorrect Fact	49%	33%	68%
Information in wrong section	2%	4%	0%
Implementation Fact Included	0%	0%	0%
Other	0%	0%	0%

In [Endres75], we have these kinds of specification or high-level design faults:

Total Specification	46%
Machine configuration and architecture	10
Dynamic behavior/communication between processes	17
Functions offered	12
Output listings and formats	3
Diagnostics	3
Performance	1

More data is given on some of the categories in the paper. They're well worth examining closely.

[Bell76] describes these kind of faults in a requirements specification. (Note: this covers only the second review described in their paper, which is more representative of a mature process.)

[1] Perhaps. [Basili87] and [Weiss85] both report on the SEL2 data, but use slightly different descriptions of the categories.

Requirement not in current baseline (a kind of added requirement)	1.5
Requirement out of scope (added requirement not in scope of contract)	7.2
Missing/incomplete/inadequate	21.0
Incorrect	34.8
Inconsistent/incompatible	9.1
New or changed requirement (Another kind of added requirement)	7.2
Requirement unclear	9.3
Typos	9.9

[Dniestrowski78] breaks down faults into phases. These are the specification faults.

	Weak complexity		Great complexity	
Fault	Totals	Percentages	Totals	Percentages
Arithmetic or Boolean Expression	0	0%	0	0%
Programming Logic	2	67%	0	0%
Software Interfaces	0	0%	0	0%
Data Declarations	1	33%	0	0%
Interrupt Handling	0	0%	0	0%
Hardware Interface	0	0%	0	0%
Documentation Standards	0	0%	0	0%
Realtime monitor interactions	0	0%	0	0%
I/O operations	0	0%	0	0%
Miscellaneous	0	0%	0	0%

[Bell76] cites a study that says as many as 12% of faults found during testing may be attributable to requirements/specification faults. The same study shows that the most common fault (8.0 - 17.8%) was missing logic -- "some logic needed as part of a successful design solution to requirements were missing", probably related to incomplete requirements.

3.2. Detailed Design Faults

In [Boehm75], design faults clustered around interfaces to the world (tapes, cards, disks, users), fault message processing, and database interface.

[Dniestrowski78] reports

	Weak complexity		Great complexity	
Fault	Totals	Percentages	Totals	Percentages
Arithmetic or Boolean Expression	4	17%	0	0%
Programming Logic	7	30%	5	15%
Software Interfaces	0	0%	1	3%
Data Declarations	1	4%	6	18%
Interrupt Handling	0	0%	5	15%
Hardware Interface	0	0%	11	32%
Documentation Standards	7	30%	4	12%
Realtime monitor interactions	1	4%	1	3%
I/O operations	1	4%	0	0%
Miscellaneous	2	9%	1	3%

3.3. Coding Faults

[Endres75] reports that, of the 38% of errors due to coding:

Total Coding	38%
Initialization	8
Addressability	7
Reference to names	7
Counting and calculating	8
Masks and comparisons	2
Estimation of range limits	1
Placing of instructions within a module / bad fixes	5

Again, there is a more detailed breakdown in the paper.

[Dniestrowski78]

	Weak complexity		Great complexity	
Fault	Totals	Percentages	Totals	Percentages
Arithmetic or Boolean Expression	1	2%	1	1%
Programming Logic	15	29%	51	65%
Software Interfaces	3	6%	7	9%
Data Declarations	17	33%	7	9%
Interrupt Handling	0	0%	2	3%
Hardware Interface	0	0%	0	0%
Documentation Standards	13	25%	5	6%
Realtime monitor interactions	0	0%	0	0%
I/O operations	0	0%	1	1%
Miscellaneous	2	4%	4	5%

(1) "Programming logic fault represents 31% of the faults for a [weak complexity] code and goes up to 50% of the faults for [great complexity] code."

(2) "Nearly 70% of the faults are introduced during the coding phase."

3.4. Testing Errors

[Herndon78] reports that 10% of errors were tester errors due to invalid test procedures.

[Ostrand84] reports that 3 of 174 change reports were generated because an independent tester mistakenly reported a bug.

4. When Faults are Detected

In [Boehm75], 54% of faults were caught during or after acceptance test. Of these, 9% were coding, 45% were design.

In [Ostrand84], we have these detection times:

Coding	5%
Unit Testing by developers	30%
Function Testing by testing group	61%
System Testing of product-product interfaces by testing group	2%

Unit testing tended to discover data handling faults (faults in changing or initializing variables). Function testing tended to discover data definition and decision problems. This is probably because the latter faults tended to be related to problems with the specification -- the programmers who tested their own code found their simple mistakes, but did not find their misunderstandings. The independent testing team did.

5. How Faults are Detected

The following table shows faults discovered either by static inspections (code reads, walkthroughs, etc.) or dynamic tests (executing the code).

Project	Static	Dynamic
[Rubey75]	44%	56%
[Schooman75]	55%	45%
[Dniestrowski78]WC	70%	30%
[Dniestrowski78]GC	50%	50%
[Weiss79]	29%	40%

NOTES:

(1) [Dniestrowski78] says, "Manual inspection detects nearly 70% of the errors for a [weak complexity] code. But the detection percentage falls to 50% for a [great complexity] code. This is consistent with the fact that the latter is very time dependent which is a difficult aspect to check by paper work analysis."

(2) The 29% [Weiss79] reports for static inspections should be considered a lower bound. Note, however, that all but one of the inspection bugs were easy to fix, whereas many more of the execution faults were hard to fix or of medium difficulty. (Question: is this because inspections found trivial bugs or because finding the bug is most of the trouble of fixing it?)

(3) 31% of [Weiss79] faults were discovered by other means, such as the Fortran compiler, preprocessor, etc. (Note: many other studies did not include such automatically-detected faults.)

(4) The following table shows how [Weiss79] inspections detected faults:

Detection method	Percent of total faults
The original programmer in "considering his program".	5%
A "quality control" inspection by someone other than the original programmer.	17%
Inspection by someone other than the original programmer for some purpose other than quality control.	5%
During translation from specifications to code	2%

(5) Of the faults detected by execution in [Weiss79], 43% involved access to particular complex data structures. Built-in error detection discovered 56% of these. (But two of the fourteen were faults associated with the error handling code itself.)

6. Refixing Bugs

[Basili81] discovered that 6% of changes were to correct or complete a previous change, 85% were fault corrections, and 9% were for other reasons.

In [Weiss85], they report that between 2.5% and 6.1% of all changes result in an fault. (Between 5% and 14% of the non-clerical faults resulted from a change.)

7. Changed Modules

[Endres75] discovered that the fault density is the same for new and modified code.

In [Basili84a], of fault containing modules (roughly, subroutines) 49% were modified and 51% were new. Both new and changed modules have a high rate of interface faults. "New modules have a equal number of errors of omission and commission and a higher percentage of control errors. Modified modules had a high percentage of errors of commission and a small percentage of errors of omission with a higher percentage of data and initialization errors. Another difference was that modified modules appeared to be more susceptible to errors due to the misunderstanding of the specifications."

[Shen85] also reports that new assembly language modules are not significantly more or less error-prone than modified modules. (New modules are significantly more error-prone than modules translated from another language.)

8. Severity of Faults

Both [Rubey75] and [Herndon78] divide faults into three categories:

Category	[Herndon78]	[Rubey75]
Low	19%	46%
Medium	50%	40%
High	31%	14%

Notes:

(1) However, they have somewhat different definitions: To [Rubey75], "Serious" faults cause crashes or "signicant" deviation from the correct values. "Moderate" faults caused lesser deviations. The end user noticed no effect because of "minor" faults. To [Herndon78], low means "minor irregularities", medium means "unsatisfactory operation", and high means that the system does not function.

(2) If you refer to the earlier [Rubey75] table, you'll see that that specification faults seem less serious than coding faults.

9. Correction Times

[Basili81] finds that inconsistency faults in a requirements specification had highest average time to fix, incorrect facts were next, omissions were next, and all others took less than an hour.

In [Weiss85], it was noted that requirements changes were not particularly troublesome, even if a project had relatively many of them. [Weiss79] discovered fewer requirements faults, but they were harder.

Both [Weiss85] and [Weiss79] report that interface faults were not especially troublesome.

[Weiss79] reports that all but one of the medium or hard difficulty faults were discovered in the last 10% of development (mostly integration testing).

[Ostrand84] reports that most problems were isolated and corrected in less than an hour. 8% of faults took more than a day to isolate; 3% of the faults took more than a day to correct. Faults found in function testing (whole system testing) were easier to isolate but harder to fix than faults found in unit testing. Specification faults were easier to isolate but harder to fix than programmer faults.

10. Faults and Modules

Both [Endres75] and [Basili84a] report that most faults caused the change of only one module. (85% for Endres; 89% for Basili.)

11. Faults and Complexity

[Basili84a] has an interesting result: a higher fault rate for small modules. It may be because of the predominance of interface faults. The same result was also noted by [Shen85], who guess that it may be because of interface faults, because more care is taken in coding large modules, and perhaps because there are more undetected faults in large modules. [Withrow90] confirmed this result for a sample of Ada programs, but also discovered that the error density rose again for modules with more than 251 lines.

In [Basili84a], the average complexity (McCabe's metric) of error-prone modules was no greater than the average complexity of the full set of modules. [Potier82] found that the McCabe metric did correlate to faults found, but only because of a correlation between size of procedure and error-proneness. When size was factored out, McCabe did not predict well. (This may also be the case in [Schneidewind79.]) Other structural measures (reachability, number of paths) correlated well, as did software science measures.

[Shen85] reports that the number of operands (variables and constants) was the best predictor of faults. This number was highly correlated to the number of decisions (branches, conjunctions, disjunctions, boolean negations, and so on). They also report that the number of faults in changed modules is better predicted by overall module metrics, rather than by metrics associated with the change alone. Perhaps many errors are caused by interactions between the fixed and changed parts.

Note: I have not made an explicit survey of work on complexity metrics. See, for example, [Basili85b] or [Kafura85].

12. Fault Type Clustering

[Potier82] identified which kinds of faults were associated with which parts of the compiler:

function	fault categories
pre-compiler	logic
lexical analyzer	interface, I/O
syntactic analyzer	interface, I/O
syntactic analyzer (2)	all categories
semantic analyzer	interface, I/O
pre-generation	interface, I/O
optimizer	interface, I/O
register allocation	data handling
optimization	computational, data base
data generation	data handling
code generation	data definition

They further break faults down into the phase in which they were made. (These are numbers, not percentages):

	requirements	spec	design	coding
computational	0	2	19	24
logic	18	70	129	81
I/O	0	3	0	4
data handling	2	17	49	54
interface	42	38	40	31
data definition	16	73	46	15
data base	0	2	5	5

Thus, these phases are associated with these faults:

Phase	categories
requirements definition	interface
functional specifications	data definition
design specifications	logic
coding	logic, data handling, database, computation

APPENDIX A

The Studies

[Boehm75] describes a a study of faults in detailed design and coding in projects that modified large existing FORTRAN programs.

[Rubey75] gives a report on faults made from over a dozen validation efforts, mainly on small commercial real-time control programs.

[Endres75] gives a description of 512 faults made in the 28th release of the DOS/VS operating system. The faults catalogued were those discovered during system test (after developer testing but before delivery).

[Shooman75] made a study of a 4K instruction control program that interfaced to other parts of the system. Faults were reported during the test and integration phase.

[Bell76] discusses a requirements document with 8248 requirements and support paragraphs in 2500 pages for a real-time ballistic missile defense system. There were 972 problems found. System was not operational at the time of the report; most problems were from reviews.

[Dniestrowski78] describes a digital flight control / avionics realtime system. 10K machine instructions, written in a mixture of LTR (a special high-level language) and assembly. Development process was waterfall: analysis, detailed design, coding and checkout, test and integration.

In [Herndon78], the application was a real-time ship-to-shore communication system. The code required 30K of storage. Faults were reported by a separate QA organization after module and module integration tests.

[Weiss79] reports on 143 faults in a project to build a 10 KLOC (excluding comments) hardware architecture simulator in Fortran. Faults were reported both during development and after delivery.

[Basili81] reports on a requirements document done as part of a redevelopment of the A-7 operational flight program. At the time of the report, the document was in the design phase. Document development took 17 person-months; 11 person-weeks were spent making the 88 changes analysed in the paper.

[Glass81] surveys 100 faults from each of two software systems for military aircraft. The first was 500K instructions and was built by 150 programmers. The second was 100K instructions and was built by 30 programmers. The faults were gleaned from post-delivery problem reports.

[Potier82] discusses faults found in a family of compilers for the high-order language LTR. More than 1000 faults were collected, the majority during the test phase, few post-delivery (maintenance). It also covers faults from [Lipow79] and [Motley77].

[Ostrand84] reports on 173 faults discovered during the development and system testing of an interactive special-purpose editor built from 10000 lines of a high-level language and 1000 lines of assembler.

[Basili84a] discusses faults in a 90,000 line Fortran project. The system, to run on an IBM 360, is a general-purpose program for satellite planning studies. It was characterized by rapidly changing requirements and by code and design reuse. Another system, ground support software where "the design is well understood and the developers have had a reasonable amount of experience with the application", is used for comparison.

[Weiss85] and [Basili87] report on projects characterized by high code reuse, established process, experienced first-level managers, and high turnover among programmers. These projects are similar to the second system of the previous paragraph. [Weiss85] reports on three systems; [Basili87] concentrates on the second of them.

[Shen85] describes post-release faults in three programs: a metrics counting tool written in Pascal, a compiler written in PL/S, and a database system written primarily in assembly languge.

[Basili85a] studied a project to redesign a satellite ground control system in Ada. There were two design levels (unit level module specifications, with algorithms, and Ada PDL). Not all of the code was tested, because no production-quality Ada compiler was ready. There was no system testing. Not all of the design was implemented. 4 people, with from 9 to 0 years of experience, participated. They did not know Ada when the project began; they had "little experience with many of the software engineering practices that Ada was designed to support". Because the project was an uncompleted training project, only a small amount of the data is reported here.

[Perry85] is a study of 94 modification requests in a 350,000 line system written in C. The study concentrated on interface faults that required modification to a global header file or more than one source file. [Perry87] extended the study to interface faults that affected a single source file.

[Withrow90] surveyed 362 Ada packages, containing 114,000 lines of code. The average package was 316 lines.

Other studies not surveyed here include [Presson81], [Mendis79], [Litecky76], [Howden81b], and more detail on TRW studies described above [Thayer78].

REFERENCES

[Amory73]
W. Amory and J.A. Clapp. *A Software Error Classification Methodology*. Technical Report MTR-2648, Vol. VII, Mitre Corporation, 1973.

[Basili81]
Victor R. Basili and David M. Weiss. ''Evaluation of a Software Requirements Document By Analysis of Change Data''. *Proceedings of the 5th International Conference on Software Engineering*, pp. 314-323, IEEE Press, 1981.

[Basili84a]
Victor R. Basili and Barry T. Perricone. ''Software Errors and Complexity: An Empirical Investigation''. *Communications of the ACM*, Vol. 27, No. 1, pp. 42-51, January, 1984.

[Basili84b]
V. Basili and D. Weiss ''A Methodology for collecting valid software engineering data''. *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 728-738, November, 1984.

[Basili85a]
V.R. Basili, E.E. Katz, N.M. Panlilio-Yap, C.L. Ramsey, and S. Chang ''Characterization of an Ada software development''. *Computer*, Vol. 18, No. 9, pp. 53-65, September, 1985.

[Basili85b]
Victor R. Basili and Richard W. Selby, Jr. ''Calculation and Use of an Environment's Characteristic Software Metric Set''. *Proceedings of the 8th International Conference on Software Engineering*, pp. 386-391, IEEE Press, 1985.

[Basili87]
Victor R. Basili and H. Dieter Rombach. ''Tailoring the Software Process to Project Goals and Environments''. *Proceedings of the 9th International Conference on Software Engineering*, pp. 345-357, IEEE Press, 1987.

[Bell76]
T.E. Bell and T.A. Thayer. ''Software Requirements: Are They Really a Problem?''. *Proceedings of the 2nd International Conference on Software Engineering*, pp. 61-68, IEEE Press, 1976.

[Boehm75]
B.W. Boehm, R.K. McClean, and D.B. Urfrig. ''Some Experience with Automated Aids to the Design of Large-Scale Reliable Software''. *Proceedings of the 1975 International Conference on Reliable Software*, in *SIGPLAN Notices*, vol. 10, No. 6, pp. 105-113, June, 1975.

[Bowen80]
J.B. Bowen. ''Standard Error Classification to Support Software Reliability''. *Proceedings AFIPS National Computer Conference*, pp. 697-705, May, 1980.

[Brown75]
J.R. Brown and M. Lipow. ''Testing for Software Reliability''. *Proceedings of the 1975 International Conference on Reliable Software*, in *SIGPLAN Notices*, vol. 10, No. 6, pp. 518-527, June, 1975.

[Dniestrowski78]
A. Dniestrowski, J.M. Guillaume, and R. Mortier. ''Software Engineering in Avionics Applications''. *Proceedings of the 3rd International Conference on Software Engineering*, pp. 124-131, IEEE Press, 1978.

[Endres75]
A. Endres. ‘‘An Analysis of Errors and Their Causes in System Programs’’. *Proceedings of the 1975 International Conference on Reliable Software*, in *SIGPLAN Notices*, vol. 10, No. 6, pp. 327-336, June, 1975.

[Fung85]
C.K-C. Fung. ‘‘A Methodology for the Collection and Evaluation of Software Error Data’’. Ph.D. dissertation, Ohio State University, 1985.

[Glass81]
Robert L. Glass. ‘‘Persistent Software Errors’’. *Transactions on Software Engineering*, vol. SE-7, No. 2, pp. 162-168, March, 1981.

[Herndon77]
M.A. Herndon and J.A. Lane. ‘‘An Approach to the Quantification of Software Errors as a Function of Module Complexity’’. *Proceedings of the 1977 Hawaii International Conference on Systems Sciences*, p. 265, 1977.

[Herndon78]
Mary Anne Herndon and Ann P. Keenan ‘‘Analysis of Error Remediation Expenditures During Validation’’. *Proceedings of the 3rd International Conference on Software Engineering*, pp. 202-206, IEEE Press, 1978.

[Howden81b]
W.E. Howden. ‘‘Errors, Design Properties, and Functional Program Tests’’, in *Computer Program Testing*, B. Chandrasakaren and S. Radicci, eds. Amsterdam, The Netherlands: North-Holland, 1981.

[IEEE83]
IEEE Standard Glossary of Software Engineering Terminology. ANSI/IEEE Std 729-1983.

[Johnson83]
W.L. Johnson, S. Draper, and E. Soloway. ‘‘An effective bug classification scheme must take the programmer into account’’. *Proceedings of the Workshop on High-Level Debugging*, Palo Alto, CA, 1983.

[Kafura85]
Dennis Kafura and James Canning ‘‘A Validation of Software Metrics Using Many Metrics and Two Resources’’. *Proceedings of the 8th International Conference on Software Engineering*, pp. 378-385, IEEE Press, 1985.

[Knuth89]
Donald Knuth. ‘‘The Errors of TeX’’. *Software Practice and Experience*. Vol. 19, No. 7, July 1989, pp. 607-686.

[Lipow79]
M. Lipow. ‘‘Prediction of Software Failures’’. *Journal of Systems and Software* , Vol. 1, No. 1, 1979, pp. 71-75.

[Litecky76]
C.R. Litecky and G.B. Davis ‘‘A Study of Errors, Error-Proneness, and Error Diagnosis in Cobol’’. *Communications of the ACM*, Vol. 19, No. **[[[]]]**, pp. 33-37, 1976.

[Mendis79]
K.S. Mendis and M.L. Gollis. ‘‘Categorizing and Predicting Errors in Software Programs’’. *Proceedings of the 2nd AIAA Computers in Aerospace Conference*. Los Angelos, October 1979, p. 300-308.

[Motley77]
R.W. Motley and W.D. Brooks. *Statistical Prediction of Programming Errors*. Technical Report RADC-TR-77-175, 1977.

[Nakagawa89]
Y. Nakagawa and S. Hanata. ‘‘An Error Complexity Model for Software Reliability Measurement’’. *Proceedings of the 11th International Conference on Software Engineering*, pp. 230-236, IEEE Press, 1989.

[Ostrand84]
Thomas J. Ostrand and Elaine J. Weyuker. ‘‘Collecting and Categorizing Software Error Data in an Industrial Environment’’. *Journal of Systems and Software*, Vol. 4, 1984, pp. 289-300.

[Perry85]
Dewayne E. Perry and W. Michael Evangelist. ‘‘An Empirical Study of Software Interface Errors’’. *Proceedings of the International Symposium on New Directions in Computing*, IEEE Computer Society, August 1985, Trondheim, Norway, pages 32-38.

[Perry87]
Dewayne E. Perry and W. Michael Evangelist. ‘‘An Empirical Study of Software Interface Faults — An Update’’. *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, January 1987, Volume II, pages 113-126.

[Potier82]
D. Potier, J.L. Albin, R. Ferreol, and A. Bilodeau. ‘‘Experiments with Computer Software Complexity and Reliability’’. *Proceedings of the 6th International Conference on Software Engineering*, pp. 94-103, IEEE Press, 1982.

[Presson81]
P.E. Presson. ‘‘A Study of Software Errors on Large Aerospace Projects’’. *Proceedings of the National Conference on Software Technology and Management*, Alexandria, VA, October 1981.

[Rubey75]
Raymond J. Rubey ‘‘Quantitative Aspects of Software Validation’’. *Proceedings of the 1975 International Conference on Reliable Software*, in *SIGPLAN Notices*, vol. 10, No. 6, pp. 246-251, June, 1975.

[Schneidewind79]
N. F. Schneidewind and H.M. Hoffman. ‘‘An experiment in software error data collection and analysis’’. *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, May 1979, pp. 276-286.

[Schooman75]
M.L. Shooman and M.I. Bolsky. ‘‘Types, Distribution, and Test and Correction Times for Programming Errors’’. *Proceedings of the 1975 International Conference on Reliable Software*, in *SIGPLAN Notices*, vol. 10, No. 6, pp. 347-357, June, 1975.

[Selby87b]
R. W. Selby. ‘‘Incorporating Metrics into a Software Environment’’. *Proceedings of the Joint Converence of the 5th National Conference on Ada Technology and Washington Ada Symposium*. pp. 326-331, ACM Ada Technical Committee, 1987.

[Shen85]
Shen, Yu, Thebaut, and Paulsen. ‘‘Identifying Error-Prone Software - An Empirical Study.’’. *Transactions on Software Engineering*, vol. SE-11, No. 4, pp. 317-323, April, 1985.

[Sukert77]
A.N. Sukert. ‘‘A multi-project comparison of software reliability models’’. *Proceedings AIAA*

Conference on Computing in Aerospace, 1977.

[Thayer76]
T.A. Thayer, et al. *Software Reliability Study*, TRW Report 76-2260.1.9-5, March 1976.

[Thayer78]
T.A. Thayer, M. Lipow, and E.C. Nelson. *Software Reliability*. TRW Series of Software Technology, Vol. 2 Amsterdam: North-Holland, 1978.

[Weiss79]
David M. Weiss. ''Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility''. *Journal of Systems and Software* , Vol. 1, No. 1, 1979, pp. 57-70.

[Weiss85]
David M. Weiss and Victor R. Basili. ''Evaluating Software development by Analysis of Changes: Some Data from the Software Engineering Laboratory''. *IEEE Transactions on Software Engineering*, vol. SE-11, No. 2, pp. 3-11, February, 1985.

[Withrow90]
Carol Withrow, ''Error Density and Size in Ada Software''. *IEEE Software*, Vol. 7, No. 1, pp. 26-30, January, 1990.

[Yu88]
T.J. Yu, B.A. Nejmeh, H.E. Dunsmore, and V.Y. Shen. ''SMDC: An Interactive Software Metrics Data Collection and Analysis System''. *Journal of Systems and Software*, Vol. 8, 1988, pp. **[[[]]]**.

[Yuen85]
C.K.S. Yuen and Chong Hok Yuen. ''An Empirical Approach to the Study of Errors in Large Software Under Maintenance''. In *Proceedings of the Conference on Software Maintenance*, IEEE Press, 1985.