An Analysis of the Fault Correction Process in a Large-Scale SDL Production Model

Dolores Zage   Wayne Zage
Software Engineering Research Center
Ball State University
Muncie, IN 47306 USA
+1 765 285 8642
{dmz, wmz }@cs.bsu.edu

Abstract

Improvements in the software development process depend on our ability to collect and analyze data drawn from various phases of the development life cycle. Our design metrics research team was presented with a large-scale SDL production model plus the accompanying problem reports that began in the requirements phase of development. The goal of this research was to identify and measure the occurrences of faults and the efficiency of their removal by development phase in order to target software development process improvement strategies. Through our analysis of the system data, the study confirms that catching faults in the phase of origin is an important goal. The faults that migrated to future phases are on average ten times more costly to repair. The study also confirms that upstream faults are the most critical faults and more importantly it identifies detailed design as the major contributor of faults, including critical faults. When the entire correction process is accounted for, this project follows the Pareto principle, or the 80/20 rule. However, when observing only downstream activities this ratio is much more extreme, approximating a 95/5 distribution.

1. Introduction

The authors have been studying design metrics as indicators of fault-prone software components in research supported by the Software Engineering Research Center. Over a fifteen-year validation period, on projects over a wide variety of application areas, these design metrics have been shown to target fault-prone design components [8,9,10,11]. The research reported in this paper is an extension of that previous metrics work with the goal of exploring the distribution of faults, their types and techniques of fault removal in the interest of providing improvements in the software development process.

The study began on a large-scale industrial software system written in the Specification and Description Language (SDL). SDL is a standard language for the specification and description of systems. Its more recent versions contain object orientation, remote procedure calls and non-determinism [5]. SDL is used in telecommunications, as well as in other real-time, distributed and communicating systems to increase productivity, reduce faults and improve maintainability.

The analysis presented here has study data that is especially useful since the data supplied was compiled as early as the requirements phase. Such thorough fault reporting is relatively uncommon and is most helpful in determining the origin and resolution of faults in the development process. The authors were fortunate enough to have been granted access to problem reports from an organization with a strong CMM[1] rating. Once the requirements had reached a satisfactory level of stability, problem reports were submitted and catalogued. This study data is also valuable because it utilizes a compressed software development environment that employs a fourth generation language with automatic code generation.

The data consisted of the production model and the related problem reports for the model. The SDL production model consisted of 1600 SDL components (processes, states, state transitions and procedures), accompanied by 602 problem reports. Our research team had the task of tracking each fault identified in the problem reports back to its SDL component. Each problem report consisted of 35 fields that included the development cycle phase of origin and phase found, severity class, a fault class, detection method and the amount of effort required to resolve the fault. In addition, a separate analysis section was appended to each report detailing the description of the problem, the problem history, the suggested cause and solution and subsequent changes to the model.

[1] CMM stands for Capability Maturity Model. This is an independent industry rating system that rates the maturity of a company's software development process. Companies are then assigned a CMM Level ranging from 1, which represents an ad hoc process to 5 representing a fully mature and evolving process. CMM Levels of 4 or above are considered to be high, and reflect development processes that are among the best in the business.

2. Analysis of problem reports

The analysis began by categorizing the problem reports by severity class and development phase. As seen from Table 1, the majority of the reports stemmed from faults attributed to detailed design. This phase also accounts for the maximum number of faults designated as critical. Of these 11 faults, 5 of them were removed in detailed design, leaving 6 to be found later in the development life cycle. Some reports have indicated that coding errors are more severe than design errors [7]. Our results indicate that detailed design in particular is the phase that contains the most critical faults in this SDL model.

Table 1. Number and severity of problem reports by development phase

	critical	major	minor	cosmetic	enhancement	unknown	total
Requirements	4	28	51	2	3		88
Preliminary Design		13	92	6	2		113
Detailed Design	11	70	135	37	1		254
Coding	2	5	3	2			12
Testing	1	4	9	3	2		19
unknown	1	14	15	2	1	84	117
total	19	134	305	52	9	84	603

For these study data, 75% (455/603) of the reports originated in the upstream phases of software development. This is a marked difference over the projects studied in the Marick Report [6]. The average percentage of the upstream faults in his survey was 39%. It has been shown [4] that faults identified earlier in the development process are cheaper to fix than faults identified late in the process. Very few (2.5%) of the total faults reported originated in the coding phase. This outcome can be attributed to the development environment, since SDL by its nature stresses design over coding. Thus design is maximized (along with the potential to originate faults) and coding is minimized. However, pushing faults into the upstream phases may reduce the total cost of the entire system. The emphasis on design in turn leads to a situation where the bulk of the faults originate in the design phases of development.

The sorting of the reports into the fifteen different fault classes is presented in Table 2. Note that the largest category, containing almost 20% of reports, was named "unknown". This category contains the reports that did not list a fault type. The two fault classes of "data" and "interface" combined contain 33% (200/603) of the reports. An interface fault interacts with other components or drivers through calls, macros, control blocks or a parameter list. Ninety of the 118 interface faults and 73 of 82 data faults entered the system during the design phase of software development.

The separation of each fault class into problems that were detected within the development phase (in-phase) and those that were detected in a later software development phase (post-phase) was completed to determine the latency for various fault types (Table 3). Observing the six classes of faults that contain over 10 reports and not including the category "unknown" (Table 2), three fault classes, namely "data", "initialize" and "reqment" roughly possess an even distribution between in-phase and post-phase discovery (Table3).

Table 2: Number and severity of problem reports by fault class

	critical	major	minor	cosmetic	enhancement	unknown	total
Constr/lim			2				2
data	2	13	39	27	1		82
err-handling		1	7				8
initialize	2	4	6	1			13
interface	2	32	74	7	3		118
logic	4	22	50	3			79
non-test			2	1			3
nonstd	2	2	1				5
other	2	15	32	2	2		53
perf/effi			1				1
rel/repeat		1					1
reqment	4	27	68	4	1		104
test - def			6	2			8
test - str		1		3			4
user - int		1	1				2
unknown	1	15	16	2	2	84	120
total	19	134	305	52	9	84	603

Only logic faults are discovered earlier: 2/3 are in-phase, 1/3 are post-phase. The two classes that exhibited the opposite pattern of discovery were the classes of "other" and "interface". Only 1/5 of the "other" reports were identified in-phase while 4/5 slipped into later phases. Interface faults also were discovered later, with 2/5 being discovered within the phase and 3/5 sliding into later phases.

When observing the remaining nine fault types, which make up approximately just 6% of all the reports (34/602), six of the nine are identified mostly in-phase and the remaining three are identified post-phase. Of these three post-phase fault types, two, “rel/repeat” and “user-int” had higher than the average effort per problem report at 5.1 (see Table 6). In fact, the single problem report denoting the fault class of “rel/repeat” expended 24 hours to remove the fault.

Table 3. Fault class separated by in and post-phase discovery

	in-phase	post-phase
constr/lim	100%	0%
data	47.0%	53%
err-handling	91%	9%
initialize	51%	49%
interface	41%	59%
logic	66%	34%
non-test	74%	26%
nonstd	74%	26%
other	29%	71%
perf/effi	0%	100%
rel/repeat	0%	100%
reqment	50%	50%
test - def	100%	0%
test - str	100%	0%
user - int	0%	100%

In a fault classification for a Hewlett Packard division system provided in Table 4, logic faults are the major source of problems, followed by documentation and computation [3]. Logic and computation are most likely to occur during coding. Our system has very few coding errors, and with its emphasis on the design phase, logic faults are minimized and removed with efficiency (see Table 3). Interesting in itself is that a category employing the concept of “other” appears in both classifications and has a significant percentage of the total faults. In this study, the faults classified as “other” have a poor in-phase discovery rate. It is well known that finding faults later magnifies the effort to correct, also verified by this study. The fault classification “other” has the second highest effort per report. Implications are that one-of-a-kind type of classifications that are placed into this catch-all category should be carefully scrutinized. Unique fault types require more effort. This statement was also observed in this study’s fault class distribution where the classes that have a small number of reports such as “rel/repeat” and “user-int” are responsible for a higher than average effort/report.

Table 4. Fault classification from a Hewlett Packard division system

Hardware	4%	Other Code	11%
Process/InterProcess	5%	Computation	18%
Requirements	5%	Documentation	19%
Data Handling	6%	Logic	32%

This historical fault classification can be useful as long as the same process and activities are executed and as long as the recording of the type of fault provides true insight into the nature of the fault. A concern is that 29% of the reports either did not record a fault class or selected the category “other”.

A second focus in this study was to measure the effectiveness and efficiency of the fault removal process. Table 5 identifies the source of the fault and phase in which that fault was identified. As observed from Table 5, 31% of the requirements faults were eliminated in the requirements phase, 30% of requirements faults were eliminated in preliminary design, 15% during detailed design and the remaining were removed during testing. When observing the large percent of coding faults removed during the testing phase, recall that there were a total of only 12 coding faults in all. However, 51% (130) of the detailed design faults slipped into the testing phase. One could say that detailed design faults had only one chance (phase) in which to be discovered, but intra-phase fault detection was 15% higher during preliminary design. It seems reasonable to conclude that practitioners could have benefited from further review during detailed design.

Unfortunately, a number of faults pass from one phase to the next as seen in Figure 1. The information in Table 5 displays the efficiency of fault removal by phase, but it does not present the cumulative effect of this slippage for the entire process. In Figure 1, the vertical arrow going into each box represents faults injected at that step. The vertical arrow going out of each box represents the faults removed at that step. The diagonal arrow represents the faults passed to the next step. As also observed from Table 5, the requirements phase has a 31% (28/88) removal efficiency. In preliminarily design, the removal efficiency increases to 57% (99/(113+60)) thus raising the cumulative efficiency to 63%. This increasing trend does not continue through detailed design. The removal efficiency decreases to 49% and the cumulative efficiency remains stable. Since there are so few coding faults, it is not reasonable to comment on that phase. Once again, as seen from the previous analysis in this paper, detailed design is the least efficient fault removal phase.

The other important aspect to efficiency is the effort required to remove the faults. If only a marginal difference exists between correcting the fault earlier than later, then many techniques can be applied at anytime for improving the fault removal process. This investigation confirms that it is costly to wait. The total effort expended to remove 236 intra-phase faults was 250.5 hours while it took 1964.8 hours to remove the 248 inter-phase faults. Faults undetected within the originating phase took approximately eight times more effort to correct. In fact, the problem doesn’t get better as time passes. “Faults found in the field are at least an order of magnitude more expensive to fix than those found while testing"[2]. This also confirms Boehm's classic result [1] that propagating faults to later

Table 5. Percentage of faults detected in each phase by phase of origin

Phase Found / **Phase of Origin**	Requirements	Preliminary Design	Detailed Design	Coding	Testing	Totals
Requirements	**31%**	**30%**	**15%**	**0%**	**24%**	**100%**
Preliminary Design		**64%**	**21%**	**0%**	**15%**	**100%**
Detailed Design			**49%**	**0%**	**51%**	**100%**
Coding				**16%**	**84%**	**100%**
Testing					**100%**	**100%**

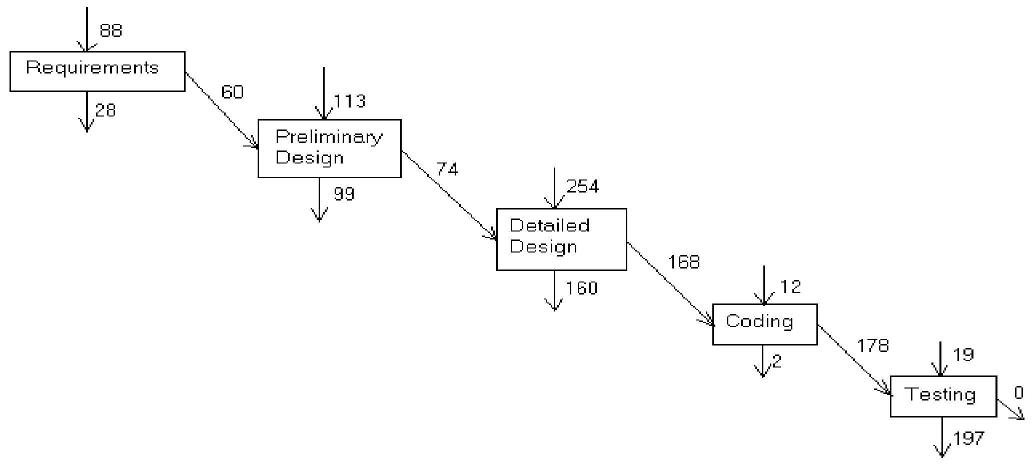


Figure 1. Profile of injected, removed and remaining faults by phase

Table 6. Percentage of effort to correct faults by phase

Phase Found / **Phase of Origin**	Requirements	Preliminary Design	Detailed Design	Coding	Testing	Totals
Requirements	**8%**	**5%**	**2%**	**0%**	**85%**	**100%**
Preliminary Design		**21%**	**12%**	**0%**	**67%**	**100%**
Detailed Design			**10%**	**0%**	**90%**	**100%**
Coding				**4%**	**96%**	**100%**
Testing					**100%**	**100%**

phases of development produces a nearly exponential increase in the effort, and thus in the cost, of fixing those faults. Table 6 quantifies the effort expended to remove faults during each phase of software development for this SDL project. From this table, one can easily determine that upstream faults that passed into the testing phase will require significantly more effort to remove than those removed earlier. To identify the effort expended compared to the volume of faults either identified in the phase of origin or passed on to subsequent phases, see Figure 2. The first column contains the requirement faults data. Removing 31% of the requirement faults in the requirements phase took 8% of the total effort. The remaining 68% of the requirement faults required 92% of the total effort to remove those faults in subsequent phases. Preliminary design, column two, had a similar relationship of faults captured within the phase of origin. However, the disparity of the effort applied to the intra and inter-preliminary design faults was not as dramatic as that of requirement analysis, but it is still significant. The third column of Figure 2 corresponds to the data of the detailed design phase. The 49% of the detailed design faults that were caught in the detailed design phase only consumed 10% of the total effort. The remaining detailed design faults consumed the lion's share, 90%, of the total effort. To place this result in another perspective, 10% of the detailed design effort was larger than the total effort expended to eliminate all of the preliminary design faults.

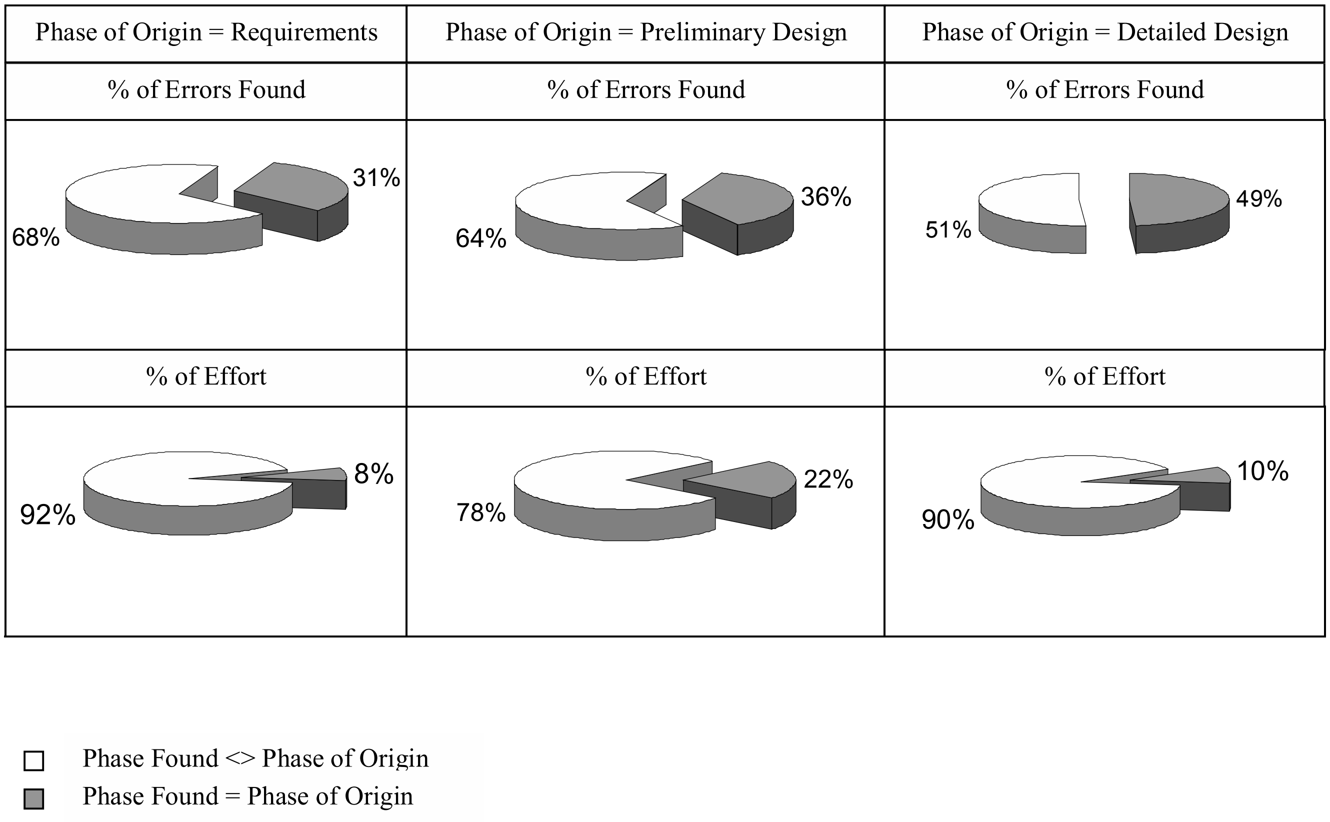


Figure 2: Analysis of inter and intra-phase faults and removal effort by development phase

The detection methods employed to uncover errors also were reviewed. The problem reports were categorized by the detection method identified in the report. There were eleven methods recorded in the problem reports as seen in Table 7. Only 3% of the reports did not record the detection method. The most expensive test detection method was integration test. On the average this detection method consumed 17.2 hours per recorded report. Note however, that approximately half of the integration test reports did not report effort. Of these reports with no effort recorded, nine were terminated and thirteen were labeled new. Additional information on the thirteen new reports could affect the average for the integration test method. However, the integration test detection method contained the highest three total efforts of the 483 reports that had recorded effort information, thus it is possible that this average will increase.

Table 7: Number and Effort by Detection Method

Detection Method	Total Number of Reports	Reports Without Effort Value	Reports With Effort Value	Total Effort	Average of Detection Method Effort for Reports With Effort Value	% of Detection Method
INTEGRATION TEST	74	36	38	652.95	17.2	12.3%
SYSTEM TEST	71	35	36	522.85	14.5	11.8%
AUTHOR CODE REVIEW	2	0	2	23	11.5	0.3%
RANDOM UNPLANNED TEST	23	5	18	202.25	11.2	3.8%
MANUAL REVIEW	92	10	82	383.25	4.7	15.3%
IN - HOUSE NORMAL USE	68	4	64	290.5	4.5	11.3%
REGRESSION TEST	22	0	22	98.75	4.5	3.6%
FUNCTIONAL TEST	56	9	47	149.75	3.2	9.3%
GROUP CODE REVIEW	5	2	3	2.25	0.8	0.8%
INTERACTIVE TEST	173	2	171	118.75	0.7	28.7%
NOT GIVEN	16	16	0	0	0.0	2.7%
CUSTOMER USE	1	1	0	0	0.0	0.2%
TOTALS	603	120	483	2444.3	5.1	100.0 %

Another observation that can be seen from Table 7 is that the detection methods of author code and group code review constitute a very minor portion of the total detection methods. Author code review appears only twice, and group code appears five times. Two conclusions come to mind: either this method was not used extensively to uncover faults or it has a very poor record in uncovering faults. However, another review category, namely manual review, uncovered 92 faults. The average effort per report using a manual review was in the top half based on effort expended of the eleven methods recorded.

The detection method of interactive test uncovered 29% of the total faults. These faults consumed 118.75 removal hours. Thus the faults that were uncovered by interactive tests were the least costly. Analyzing the 173 problem reports listing interactive test as the detection method, 141 of the faults were discovered within the development phase, while the 32 remaining faults were from previous phases. The source for 89 of these interactive reports was test problems. Perhaps the reason that interactive tests uncovered problems with less effort is that this method uncovered only the least severe problems. When the reported severity classes for the reports listing the interactive test as the detection method were compiled, it was found that 52% of all the cosmetic and 39% of all the minor class severity reports were identified by interactive tests. Interactive test has a greater share of minor and cosmetic faults than the other 11 detection method categories, thus it should have the greatest efficiency per test. However, this method was also responsible for uncovering four critical problems while only consuming less than 4 hours effort. Only in-house normal use and system test uncovered as many critical faults, but each method required additional effort at 11 hours for in-house and over 65 hours each for system test.

It is also interesting that reports listing critical faults do not consume the most effort, but the reports cataloging enhancements and major faults consume the most per enhancement or fault. These data can be seen in Table 8. The majority of the reports designate the severity level as minor or cosmetic. Sixty-two percent of the reports recorded less than one hour of effort to correct the problem. These 62% accounted for only 7% of the total effort. There are only 29 reports with effort recorded at 20 hours or more, and yet they consume 1380 hours or 56% of the total effort for all of the reports listing. Or stating it as a distribution 5% of the reports took 56% of the effort. Seven of the twenty-nine reports listed requirements as the phase of origin. Two reports in this set listed preliminary design as the phase of origin. Eighteen reports listed detail design.

The phases of coding and integration testing each listed one. As previously noted, the phase of detail design has the most reports identifying the severity level as critical, it is also the phase of origin of the most time consuming reports to fix.

Table 8: Severity Classes Total and Effort

Severity Class	Number	Effort	AVERAGE EFFORT PER REPORT
Critical	19	117	6.2
Major	134	1103.7	8.2
Minor	305	1005.1	3.3
Cosmetic	52	66	1.3
Enhance ment	9	152.5	16.9
Unknown	84		0
Total	**603**	**2444.3**	**4.1**

All of the 29 reports were uncovered during later stages of development. The majority of them were uncovered during integration (8) and system testing (12) phase.

The fault class of these 29 reports was also explored. Fourteen of the reports listed interface as the fault class. From Table 2 the percentage of interface faults for the entire set of reports falls at about 20%. When considering only the 29 reports with effort greater than 20 hours, interface fault now constitutes 48%. As expected, the other fault classes of data, logic, and other roughly approximate the same proportion as in the total report population. Requirement faults have less than the expected distribution at 2 reports. This perhaps can be explained by the fact that all of these reports stem from the later stages of development and the majority, if not all of requirement fixes have had time to resolve themselves. Surprisingly the fault classes with very few reports in the total report set such as err-handling, initialize, rel/repeat are represented within these 29 reports. This suggests that uniqueness has its price in terms of effort.

The problem reports also contain documentation on the history of the report. For example, the submission date, the submitter, and the resolve date are part of the information. Also as part of this information is whether a problem was cloned from another problem set, for example the associated hardware problem reporting. Of the 29 reports, 16 (55%) were cloned. In the entire database of 603 reports, there were 62 reports that were cloned. Of these 62 reports only 24 had recorded an effort detection method of which 16 took over 20 hours to correct.

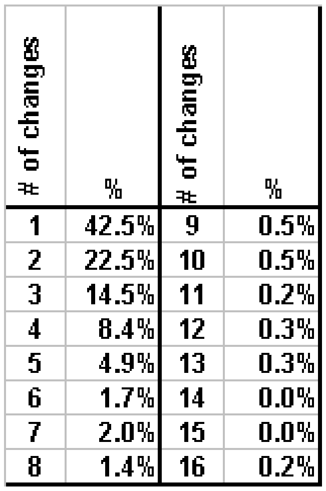
The detection methods by each development phase were analyzed to determine which methods are most likely to uncover certain fault categories. This analysis did not uncover any surprising results. The detection methods divided themselves more by their availability to perform the test. For example, in-house normal use was the best at identifying problems with requirements. Interactive test was the best method in finding preliminary design and detail design problems. Random unplanned test and system test also discovered requirement problems, but these required more than thirty hours each on average.

The detection methods categorized by fault class were compiled. There were 11 detections methods. The top three fault classes, data, requirement and interface faults, which made up over 50% of the reports were isolated. The distribution of the type of methods used to uncover all of the various fault classes exhibited the same pattern of discovery methods for the individual fault classes of data, requirement, and interface. This implies that even if a tester has the knowledge of the fault class, the detection methods applied would be similar to when this information is not available.

3. Analysis of Model Changes

When the changes for each problem report were distributed to approximately 1600 modules in the SDL production system, 37% of these modules had changes while 63% did not. Also recall that these problem reports began with the requirements phase and ended with maintenance, thus these changes are the aggregate changes for the entire development process over the entire model. As expected from the modules that had changes (Table 9), most of them only had one change. The greatest number of changes for a module was 16. This particular module was a setup module and even with its sixteen changes, the effort associated with these sixteen changes was a little over 26 hours.

Table 9: Percentage of modules with 1 to 16 changes



# of changes	%	# of changes	%
1	42.5%	9	0.5%
2	22.5%	10	0.5%
3	14.5%	11	0.2%
4	8.4%	12	0.3%
5	4.9%	13	0.3%
6	1.7%	14	0.0%
7	2.0%	15	0.0%
8	1.4%	16	0.2%

The modules that required 2 changes were isolated to determine if fault types occur in pairs. Since there were 15 fault classes, pairs of faults can occur in a total of 105 fault class combinations. (15*14)/2. When analyzing the 150 modules that required 2 changes, only 19 combinations were found. and three of the pairs occurred 12%,13% and 14% of the total 105 two change modules.

4. Conclusions and future research directions

The results indicate that detailed design is the phase that introduces the most critical faults in a developing system. Detailed design is also the least efficient fault-removal phase. Moreover, this investigation confirms that it is very costly to remove faults if they are not detected in their phase of origin. The types of fault-detection methods used to uncover the various fault classes exhibit the same pattern of discovery, suggesting that even if a tester has the knowledge of the fault class, the detection methods applied would be similar to when this information is not available. The results of this study suggest that a further analysis of a system during detailed design is needed to capture a relatively high number of faults that often slip through to downstream life cycle phases.

In the future, a detailed analysis of the module changes will be performed in order to offer further guidance during testing. In addition, we will explore whether the functionality of the module can be used to identify appropriate fault-detection techniques.

5. Acknowledgements

The research presented in this paper was funded by Motorola Corporation through the Software Engineering Research Center (SERC), a National Science Foundation Industry/University Cooperative Research Center (NSF Grant No. ECD-8913133).

6. References

1. Boehm, B. W. “Software Engineering”, *IEEE Transactions on Computers,* December 1976, pp. 1226-1241.

2. Dalal, S.R., J.R. Horgan, and J.R. Kettenring, “Reliable Software and Communication: Software Quality, Reliability, and Safety”, *Proceedings of the 15th International Conference on Software Engineering,* 1993, pp. 425-435.

3. Daran, M., and P. Thévenod-Fosse, “Software Error Analysis: A Real Case Study Involving Real Faults and Mutations”, *ISSTA ’96*, pp. 158-171.

4. Eick, S. G., et al, “Estimating Software Fault Content Before Coding”, *Proceedings of the 14th International Conference on Software Engineering*, 1992, pp. 59-65.

5. Ellsberger, J., D. Hogrefe, and A Sarma, *SDL, Formal Object-oriented Language for Communication Systems*, Prentice Hall, Europe, 1997.

6. Marick, B. “A Survey of Software Fault Surveys”, Report No. UIUCDCS-R-90-1651, Dept. of Computer Science. U. Illinois at Urbana-Champaign, December 1990.

7. Rubey, R.J., Quantitative Aspects of Software Validation, *Proceedings of the 1975 International Conference on Reliable Software - SIGPLAN Notices,* Vol. 10, June 1975, pp. 246-251.

8. Wong, W.E., J.R. Horgan, W. Zage, D. Zage and M. Syring, "Applying Design Metrics to Predict Fault Proneness: A Case Study on a Large Scale Software System," *Software-Practice and Experience*, Vol. 30, No. 14, November 25, 2000, pp. 1587-1608.

9. Zage, W., D. Zage, J. M. McGrew, and N. Sood, “Using Design Metrics to Identify Error-Prone Components in SDL Designs”, *Proceedings of the 9th International SDL Forum*, Montreal, Canada, June 1999.

10. Zage, W., D. Zage, E. Wong, J. R. Horgan, and M. Syring, “Applying Design Metrics to a Large-Scale Software System”, *Proceedings of the Ninth International Symposium on Software Reliability Engineering (ISSRE ’98)*, Paderborn, Germany, November 1998.

11. Zage, W. and D. Zage, “Evaluating Design Metrics on Large-Scale Software”, *IEEE Software Journal,* July 1993, pp. 75-81.