

Quantitative Analysis of Development Defects to Guide Testing: A Case Study

C. Stringfellow, A. Andrews*
Computer Science Department
Colorado State University
Fort Collins, CO 80523
aaa@cs.colostate.edu, stringfellow@cs.nmhu.edu
970-491-7016(ph) 970-491-2466(fx)

Abstract

Many quality improvement activities can be guided by defect analysis. Development defect analysis of software components can be used to guide testing with the goal of focusing on parts of the software that were fault-prone during development. We perform a case study using defect data from a large software product (medical record system). In this study, development defect data help to identify which parts of the software should be tested more and earlier because they were fault-prone during development. Several testing guidelines are proposed to make system test more effective and more efficient.

1 Introduction

Many attributes of a software project influence the software testing effort, both its effectiveness and efficiency. Example attributes are complexity of the problem, schedule urgency, and the quality of work during design and implementation. We are particularly interested whether defects found during development are related to the occurrence of defects during testing. The rationale for such a relationship is that:

- Components with severe or systemic problems during development carry higher risk of not being completely fixed at the start of system test.
- They are more likely to exhibit long term problems.
- These problems are more likely to be severe.
- If this is the case, defect data from development can guide testing.

Specifically, components that are fault-prone during development should be system tested at the earliest possible time. This would shift higher defect intensities earlier in the test cycle, giving developers more time to fix remaining problems.

Combined with a statistical stopping rule [8, 13] this could also make it possible to finish testing earlier: Cumulative defect curves have fewer spikes late in the testing phase with such rules and allow for earlier stopping points. Whether or not it is feasible to reorder testing depends on several factors. Components may have to be tested in a specific order, if they depend on other components working correctly. This problem can be addressed by a “qualification” phase before system test which determines readiness for

*corresponding author

system test. Another factor concerns how testing is performed. System testing may perform operational testing; it may test against specifications; or it may test features and components. Only if defective components are identified in defect reports can reordering take place. Finally, if testing cannot be flexible, this approach is not possible.

An important issue for testers is to what degree they prevent post release defects and which types of components make it into release with undetected defects that show up after release. To this end, we performed a case study in which we analyzed defect reports of three releases of a large medical record system. We used Ohlsson et al. [12] to identify fault-prone components within and across releases. We determined the quality of predicting fault-prone components during testing from development defects and evaluated the effect of reordering testing efforts. We also analyzed post release defects.

Section 2 presents approaches which have been used to identify fault prone components with the goal of using that information to make testing more effective. Section 3 describes our approach. We illustrate it on defect data of three releases of a large medical record system. We also evaluate the effect of the testing guidelines on efficiency improvement. Section 4 draws conclusions.

2 Background

Frankl, et al. [5] see two main goals in testing: to achieve quality, by probing the software for defects that can be removed, and to assess quality to gain confidence that software is reliable. Depending on the goal, different testing strategies are used - debug testing to achieve quality and operational testing to assess quality. The testing strategy used also determines the kinds of defects found. Debug testing focuses on finding defects with a higher probability of being detected, not necessarily those most important, while operational testing focuses on finding defects that are most likely to occur in the field, with probabilities in proportion to their use. Their study found that when testing software for the purpose of improving quality (i.e., removing defects), it helps if the tester has good intuition and insight in devising testing strategies, otherwise operational testing is indicated. Aids to improve a tester's intuition or validate it would be helpful. Knowing which components are fault-prone allows the tester to concentrate on devising testing strategies for those components.

Much of the research in the area of defect analysis on software components has focused on identifying fault-prone components [6, 7, 10, 11, 14] or predicting the number of defects remaining in components [2, 3, 16, 17] based on their characteristics. It is useful to know which components are fault-prone during prior release or in earlier life cycle activities, as these components should be tested more intensely. Data from previous releases can be used, but if this data is not available, data from earlier life cycle phases may be useful.

Ash et al. [1] provide a method to track fault-prone components across releases. Schneidewind [14], Khoshgoftaar et al. [6] provide methods to predict whether a component will be fault-prone. Other methods [11, 12, 15] combine prediction of fault-prone components with code decay analysis by looking at relationships between components. Ohlsson, et al [11] rank components based on the number of defects written against a component: Ranks and changes in ranks are used to classify components as green, yellow or red (GYR) over a series of releases. Red components are the most fault-prone.

Biyani, et al. [2] explore the relationship of the number of faults per module to the prior history of a module, specifically, they use faults discovered in development to predict faults remaining in the field. They also use faults discovered in previous releases to predict faults in the current release. Their method compares the number of defect-free modules in the field to the number of defects found in development, as well as the average number of defects found in the field per module to the average number of defects found in development per module. They found that modules with more development defects tended to have more field defects. In addition, comparing defects in a current release to those in previous releases, they

concluded that the prior release was sufficient for predicting the number of defects during development or in the field. In their study, defect data from development and data from the prior releases are good measures for assessing the relative quality of software.

Fenton and Ohlsson [4] test several software engineering hypotheses, some of which relate to the Pareto principle of distribution of faults, the use of early fault data to predict later fault data, and the use of metrics for fault prediction. They discovered there is evidence to support the Pareto principle that a small number of modules contain most of the fault discovered. They found no evidence to support the hypothesis that module size or complexity metrics are a good predictor of fault-prone modules. They did find strong evidence that modules that are fault-prone in pre-release are least fault-prone in post release and module that are most fault-prone post release are least fault-prone pre-release.

Unlike Ohlsson et al. [12], our method identifies and analyzes fault-prone components not only across releases, but across development and test as well. As in Fenton et al. [4], we found that a module with a higher incidence of faults in development had a higher incidence of faults in system testing. Our purpose is to use this information to guide testing, rather than tracking components across releases or looking at relationships between components to analyze code decay. Unlike Biyani, et al. [2], we are interested not so much in predicting the number of defects between development and post release and across releases, but in how we can use defect data to improve system testing, both its effectiveness and efficiency. In this we combine various aspects of assessment and prediction, and use the data to support suggested improvement activities.

3 Data

We analyzed defect reports for three releases of a large medical record system. All releases added functionality to the product. The software consists of 188 software components. (A component is a set of files, that are logically or physically related.) Of the 188 components, 100 had at least one defect report in release 1, 2 or 3. Seven new components were added in release 1, five new components were added in release 2, and three new components were added in release 3. In addition, one of the new components in release 1 saw major enhancements in release 2. Many other components were modified in all three releases as a side effect of the added functionality.

Developers, system testers and users (after release) report problems encountered by submitting defect reports. The software project database records the following attributes for defects reports:

- release identifier
- phase in which defect was reported (development, test, post release)
- defective entity (code component(s), type of document by subsystem)
- whether the component was new for a release
- the date of the report
- the status of the defect (valid or invalid). A defect is valid if it is repeatable, does cause the software to execute in a manner not consistent with the system specifications, and has not already been reported.
- the “drop” in which the defect occurred. Software is released to testers in stages or “drops”. Developers work on drop “i+1” when testers test drop “i”. Each successive drop includes more of the functionality expected for a given release.

We refer to valid defect reports as defects throughout this paper. Only valid defect reports are considered in the analyses.

4 Approach

While many prior studies have performed controlled experiments, we performed a case study using data from a commercial application to show how these methods can be applied in practice. The approach used in this case study consists of the following stages:

1. Identify whether fault-prone components during development are a good predictor of fault-prone components during test and derive a test guideline for this situation.
2. If prediction has false positives and false negatives, determine whether other attributes of components or defects would improve this prediction. Examples of such attributes are whether a component is new or how severe a defect is. We purposely restrict ourselves to data that is usually available in a defect database. We derive a test guideline for this situation.
3. Evaluate effects of fault-prone components on post-release problems. Derive additional test guidelines.
4. Summarize applicable test guidelines and evaluate how well they work on the second and third releases.
5. Perform cross release analysis.

4.1 Fault-prone Component Analysis

Components are ranked by the number of defects found in them during development and system test. Ohlsson, et al. [11] used such a defect ranking on overall defects in successive releases to identify components that are fault-prone across releases. The purpose was to identify possible code decay. By contrast, we use this defect ranking to identify components that are fault-prone during development versus those that are fault-prone during system test.

Similar to [11] we consider a component “red”, if it is fault prone in both development and test. It is “green”, if it is fault prone in neither development nor test. Finally, a component is “yellow”, if it is fault-prone in either development or test, but not both. Testing should focus on the “red” components. They should be tested as early and as thoroughly as possible.

Setting the threshold for the number of defects that makes a component fault-prone can be done in two ways:

- as a percentage of the ranked components (e.g., the top 25%)
- as a function of the total number of defects.

Both methods were investigated in this case study. A fairly high threshold is needed to avoid a large number of components that are fault-prone in either development or test, but not both. The goal is to reduce the number of “false positives” and “false negatives” when using development fault-prone classification for prediction.

How to set the threshold may depend on the expectations for quality of the system and the amount of resources available for extra testing. We set the threshold based on how many components had defects during development. For the project analyzed, a threshold of 25% is too much. It requires defects to be found in 45 components. Table 1 shows that in release 1 system test only had 32 components with a reported defect. Development found defects in exactly 45 components, and many of these components only had one defect. A component having one reported defect arguably is not fault-prone. The percentage needs to be set lower. We explored 10% and 5%.

	Release 1 (180 components)		Release 2 (185 components)		Release 3 (188 components)	
	development	test	development	test	development	test
Total number of defects	460	177	299	204	19	77
# components with a defect	45	32	41	39	11	19
Mean # defects per component	2.56	0.98	1.62	1.10	0.10	0.41
Std. Dev.	12.38	3.74	9.11	3.67	0.59	3.12

Table 1: Statistics on defects found by development and system test.

Tables 2 and 3 show diffusion matrices for release 1 using 10% and 5% of the ranked components as a threshold. A diffusion matrix shows the number of components that are fault-prone or normal during development and during system test. Chi-square tests on the data in Tables 2 and 3 show significant differences ($p \leq 0.001$) between components that were fault-prone and components that were normal in development. Table 2 shows this analysis using a threshold of 10% ($\chi^2 = 46.1180$; $p \leq 0.001$). Table 3 shows this analysis using a threshold of 5% ($\chi^2 = 105.6387$; $p \leq 0.001$). Table 2 (10% threshold) shows that 10 of the 18 components identified as fault prone in development stayed that way in system test (top left cell). Of these 10, 5 were new components. The other 8 are normal in system test (top right cell). Using development fault-prone components as a predictor of fault-proneness during test would classify these eight components incorrectly. If the test guideline stated to test development fault-prone components more thoroughly during system test, this would cause system test to overtest these 8 components. In addition, release 1 had 8 fault-prone components that were normal (not fault-prone) in development. Had the test guidelines stated to test components less intensively, if they were not fault-prone during development, these components would not be tested enough and test could miss defects.

Table 3 (5% threshold) shows better results: Fewer components are overtested and fewer fault-prone components are missed. If a system is of high quality, a smaller threshold should be chosen. To summarize, setting the threshold should be guided by quality expectations about the system.

Threshold = 10% 18 components	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	10 (5 new)	8 (1 new)
Development Normal	8 (2 new)	154
Chi-square Results	$\chi^2 = 46.1180$; $p \leq 0.001$	

Table 2: Diffusion Matrix for Release 1 using top 10% of ranked components as threshold.

The threshold may also be set as a function of the number of defects. Table 4 shows the highest number of defects written against a single component in development and test for each release. We set the threshold at about an order of magnitude less than the largest number of defects written against a component during development. For release 1, the maximum number of defects for a single component in development is 143. The threshold was set at an order of magnitude lower, at 10 defects per component. This threshold applied to both development and system test. Table 4 shows the number of components that were over this threshold and considered fault-prone and the corresponding percentage of components that were fault-prone.

Threshold = 5% 9 components	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	7 (5 new)	2 (1 new)
Development Normal	2 (1 new)	169
Chi-square Results	$\chi^2 = 105.6387$; $p \leq 0.001$	

Table 3: Diffusion Matrix for Release 1 using top 5% of ranked components as threshold.

	Release 1 (180 components)		Release 2 (185 components)		Release 3 (188 components)	
	development	test	development	test	development	test
Max defects in a component	143	27	115	32	7	41
# components over threshold (10 defects for releases 1&2) (4 defects for release 3)	10	6	6	5	1	3
% components over threshold	5.6%	3.3%	3.2%	2.7%	0.5%	1.6%
Mean # defects for components over threshold	36.06	19.33	35.17	20	7	19.33
Std. Dev. over threshold	40.57	6.12	39.97	7.38	0	18.82

Table 4: Statistics on fault-prone components using order of magnitude threshold.

In release 1, a significant difference exists between fault-prone and normal components ($\chi^2 = 71.5619$; $p \leq 0.001$) using an order of magnitude threshold. Ten components were fault-prone in development, six were fault-prone during system test. Table 5 shows to what degree a component that is fault-prone (or normal) during development stays that way during system test. If one were to predict fault-proneness dur-

Threshold ≥ 10 defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	5 (4 new)	5 (1 new)
Development Normal	1 (1 new)	169 (1 new)
Chi-square Results	$\chi^2 = 71.5619$; $p \leq 0.001$	

Table 5: Diffusion matrix for Release 1 using order of magnitude as a threshold.

ing testing based on whether a component was fault-prone during development, the diffusion matrix can be used to evaluate the quality of the prediction model. In this case, it correctly predicts five components as fault-prone during system test and 169 components as normal during system test. It shows five false positives (development fault-prone, but normal during system test) and one false negative (normal during development, but fault-prone during system test). If one were to use the classification of fault-prone components during development as a guide to increase testing effort for fault-prone components and to decrease test effort for normal components, this would lead to “overtesting” five components (an inefficiency) and not testing one component enough (a decrease in effectiveness). This is not quite the accuracy desired.

In this case study, using order of magnitude to determine threshold has results similar to the threshold set at 5%. This is because both result in roughly the same number of components. Using an order of magnitude difference in number of defects for a component also included a component that was consistently fault-prone in later phases (system test and post-release) of release 1 and throughout all phases (development, system test, and post-release) in release 2. A threshold set at 5% would have missed this fault-prone component in release 1. Because the quality of a system may not be known prior to system test, it would be difficult to set the threshold based on a percentage of ranked components. Therefore, setting the threshold as a function of the maximum number of defects found in a single component is recommended.

A testing guideline that emphasizes more thorough testing of components that were fault-prone during development would work well, catching components that are fault-prone in system test, and overtesting a few. A guideline that de-emphasizes testing of components that are not fault-prone during development would generally work well and potentially save a lot of effort, since the vast majority of components fall into that category.

4.2 Consideration of Other Indicators

Table 5 also shows how many of the components in each category were new for release 1. Interestingly, all but one were fault-prone in either development or test, and thus could benefit from more thorough testing. This led to a second testing guideline, based on the assumption that components that are development fault-prone or new will be fault-prone during system test and should be tested more thoroughly. This guideline removes the false negative in the lower left cell of the diffusion matrix of Table 5 by grouping it correctly with the components that are fault-prone during system test. We still over-test components that are fault-prone during development, but are not fault-prone during system test. However, this guideline no longer misses the new component that is not fault-prone during development, but fault-prone during system test. Table 6 shows the results of this analysis. A chi-square test on the data in the table shows a significant difference ($\chi^2 = 86.8966$; $p \leq 0.001$) between components that were fault-prone in development or new versus components that were normal in development.

Threshold ≥ 10 defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	6 (5 new)	6 (2 new)
Development Normal	0	168
Chi-square Results	$\chi^2 = 86.8966$; $p \leq 0.001$	

Table 6: Diffusion Matrix including new components in fault-prone category in Release 1.

Another possible refinement of test guideline 1 is to refine it with respect to defect severity level. Severity levels range from 1 to 4. Level 1 is the highest. Table 7 shows the maximum number of defects found during development and system test for a component for each severity level.

Because the maximum number of defects in each severity level is lower than the cumulative number of defects, we need to set a new threshold. The results of the fault-prone analysis between development and testing for release 1 are presented in Tables 8 - 11. Chi-square analysis did not show a significant difference between fault-prone and normal components in development using severity levels 1, 2, and 4 at $p \leq 0.001$. The analysis does not indicate any benefits in using severity levels for prediction of fault-prone components. Analyzing defects by severity level does not improve the predictions. Thus testing guidelines based on severity level would not improve the process.

	Severity Level			
	1	2	3	4
Development	54	47	27	15
System Test	5	17	21	6

Table 7: Maximum defects for a component by severity level for Release 1.

Threshold ≥ 4 defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	1 (1 new)	10 (5 new)
Development Normal	1 (1 new)	168
Chi-square Results	$\chi^2 = 6.7898; p \leq 0.001$	

Table 8: Diffusion Matrix for Release 1 by severity 1.

Threshold ≥ 4 defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	1	6 (3 new)
Development Normal	3 (2 new)	170 (2 new)
Chi-square Results	$\chi^2 = 4.8780; p \leq 0.05$	

Table 9: Diffusion Matrix for Release 1 by severity 2.

Threshold ≥ 4 defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	4 (4 new)	3
Development Normal	3 (2 new)	170 (1 new)
Chi-square Results	$\chi^2 = 55.2623; p \leq 0.001$	

Table 10: Diffusion Matrix for Release 1 by severity 3.

Threshold ≥ 4 defects	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	0	3 (2 new)
Development Normal	2 (2 new)	175 (3 new)
Chi-square Results	$\chi^2 = 0.0343; p \leq 1$	

Table 11: Diffusion Matrix for Release 1 by severity 4.

4.3 Comparison to fault-prone components in post release

The next analysis questions were: Is system test finding all problems? What is the nature of the problems that are first detected in the field? To investigate this we compared fault-proneness in system test to post-release fault problems. Using the order of magnitude approach, the fault-prone threshold was two post-release defects per component. This identified seven components as fault-prone. Table 12 shows the results. Chi-square analysis shows no significant differences ($\chi^2 = 2.7114$; $p \leq 0.10$) exist between components that were fault-prone in system test versus components that were normal in system test after release.

Post release threshold ≥ 2 defects System Test threshold ≥ 10 defects	Prediction (Post Release)	
	Fault-prone	Normal
System Test Fault-prone	1	5 (5 new)
System Test Normal	6	168 (2 new)
Chi-square Results	$\chi^2 = 2.7114$; $p \leq 0.10$	

Table 12: Diffusion Matrix for System Test versus Post-Release in Release 1.

Only one component was fault-prone in both system test and post release (it was also fault-prone in development). It is not a new component and easily identifiable by the high number of defects throughout the life cycle. Obviously, such a brittle component is cause for concern.

Five components were fault-prone during system test, but were repaired before release and problem-free (all of them new components). This indicates thorough testing and excellent repair work for new components.

Harder to identify before release are the components that are normal during system test, but fault-prone after release. None of them are new, but may have been affected by changes or enhancements. One possible cause for missing these components may be that regression test didn't test them thoroughly enough. The defect data is not detailed enough to explain this phenomenon. A suggestion for improving testing would be to assess (and possibly improve) change impact analysis and regression testing.

Overall, development and system test are doing a very good job of testing and repairing new components. The exposure rate of defects for new components is high and these components have very few defects after release indicating that the problems were corrected. Within release analysis therefore will not help guide system test in preventing post release defects.

4.4 Reordering System Test Activities

Besides using identification of fault-prone components during development to guide system test as to which components should be tested more thoroughly, knowledge of fault-prone components can also help in determining which components could benefit from being tested earlier. Whether or not it is feasible to reorder testing activities depends on several factors. Components may have to be tested in a specific order, if they depend on other components working correctly. In such a situation, reordering may not be possible. However, in this case study, the software development process contains a "qualification" phase before system test which determines test readiness. The system test group runs a subset of the test cases they have developed. If the software has major problems, it is sent back to development. This means that when system test starts, components do not have to be tested in a specific order. In addition, system test identifies and schedules features and major components for testing, as well as doing regression testing. Therefore, it is possible to rearrange the schedule for testing specific components.

Testing guideline 3 is to test components that are fault-prone during development as early in a drop as possible. Earlier identification of defects gives development more time and flexibility for correction. Components that were not fault-prone should be tested late in the test period for a given drop.

To evaluate the effect of this guideline, we identified the weeks in system test which tested components that were fault-prone during development. To simulate testing these fault-prone components earlier, test results of those weeks were switched with those of an earlier test week (during the same drop) during which non-fault-prone components were tested. Release 1 has three drops in system test. Testing on drop 1 started during week 11, on drop 2 during week 20, and on drop 3 during week 27. Figure 1 shows cumulative defects per week for the original test process, and for the one that tests components that were fault-prone during development as early in a drop as possible. The reordered curve clearly indicates earlier

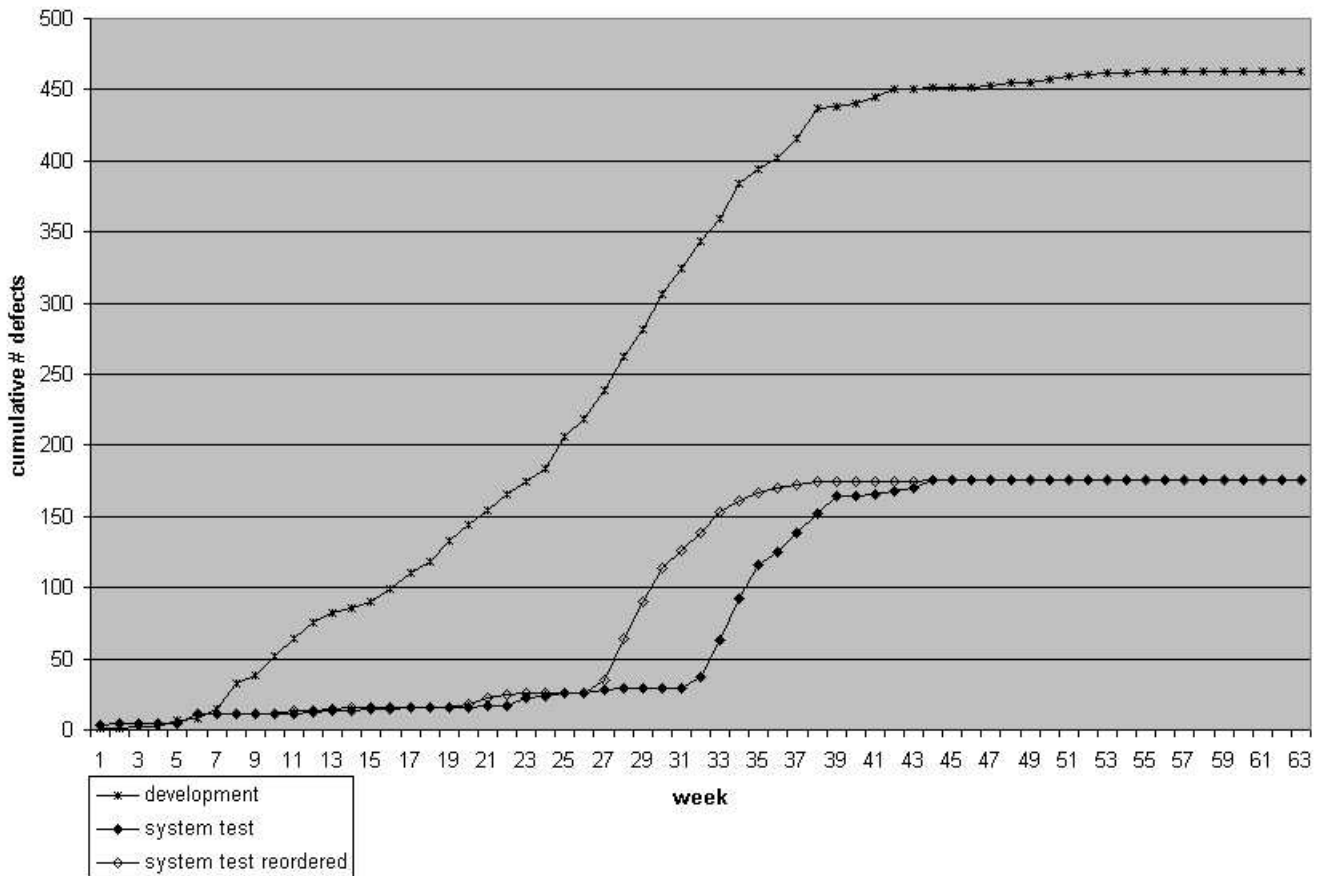


Figure 1: Cumulative defects for development, system test and reordered system test in release 1.

discovery of defects. It also is smoother, especially at the end, because defect spikes caused by late testing of fault-prone components have disappeared. Beyond the benefit of earlier defect identification, applying a statistical stopping rule [9, 13] to the cumulative defect curve will allow for earlier stopping than in the original curve, reducing the elapsed time needed for testing. Depending on the stopping rule, this could save 6-8 weeks for release 1. The approximate cost per week of testing is about \$100K. Thus this testing guideline could easily have saved \$600 - 800K. This is a substantial amount of money.

4.5 Analysis of Release 2 and Release 3

To fine tune the testing process, one should develop guidelines that are specific in a project group and are based on project data from that environment. Data from release 1 was used to derive the guidelines. Table 13 shows these guidelines. There were also several candidate guidelines that did not work well in this

Guidelines	
1.	Test development fault-prone components more thoroughly. (Test components not fault-prone in development less.)
2.	Test new and development fault-prone components more thoroughly.
3.	Test development fault-prone components as early in a drop as possible.

Table 13: Guidelines for testing

environment. They related to defect severity levels and the quality of the repair work between development and system test, and before release. To evaluate whether the test guidelines are useful, they were applied and evaluated on release 2 and release 3 of the same system.

In release 2, the maximum number of defects per component found in development is 115. In system test it is 32. We chose the same threshold (10) as in release 1. Any component with more than ten defects was considered fault-prone.

In release 2, six components were identified as fault-prone during development. Only two of them were also fault-prone during system test. Applying guideline 1 would have over-tested the other four. There were five fault-prone components in system test, three of which were not fault-prone during development and would not have been tested enough according to guideline 1 (cf. Table 14). This latter misclassification is more dangerous.

Threshold ≥ 10	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	2 (1 new)	4
Development Normal	3 (2 new)	176 (2 new)
Chi-square Results	$\chi^2 = 22.1250; p \leq 0.001$	

Table 14: Diffusion Matrix for Release 2.

Applying the more conservative guideline 2 improves matters. The chi-square test results improve when new components are grouped with development fault-prone components (from $\chi^2 = 22.1250$ to $\chi^2 = 91.7661; p \leq 0.001$). Table 15 shows the results. While we “over-test” more components than with the first guideline, we only undertest one (old) component that is fault-prone during system test.

As for release 1, defect data analyzed for severity do not give better results and do not warrant additional test guidelines.

Next, we evaluated whether system test defects are a good indicator of post release defects. Table 16 shows there was no significant difference ($\chi^2 = 0.1427; p \leq 1$). The results quite strongly point out that any component that was fault-prone during test was **not** fault-prone after release. This is good news: the problems that were identified during testing were corrected before release. Development and system testing are doing a very good job of eliminating problems. Improvements in testing should focus on the five components that were not fault-prone during development or test, but showed post release problems.

Threshold ≥ 10	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	6 (5 new)	4
Development Normal	1	174
Chi-square Results	$\chi^2 = 91.7661$; $p \leq 0.001$	

Table 15: Diffusion Matrix including new components in fault-prone category in Release 2.

These components were not new. One possibility is to investigate whether the problems could have been prevented with more extensive change impact analysis and regression testing. This requires more detailed data than we had available.

Post Release Threshold ≥ 2 System Test Threshold ≥ 10	Prediction (Post Release)	
	Fault-prone	Normal
System Test Fault-prone	0	5 (1 new)
System Test Normal	5	175 (4 new)
Chi-square Results	$\chi^2 = 0.1427$; $p \leq 1$	

Table 16: Diffusion Matrix for System Test versus Post Release in Release 2.

Test guideline 3 specifies that components that were fault-prone during development should be tested as early as possible in a drop. As in release 1, we evaluated the effect of this guideline by shifting testing of development fault prone components to an earlier week in the same drop. Figure 2 shows the results.

As in release 1, defect intensity is higher earlier in testing. Most defects are found earlier, as well, giving developers more weeks to fix problems. As before, stopping rules are also likely to identify earlier end points for testing, saving effort and calendar time for the testers. However, there were fewer defects in release 2 overall, and thus the benefits of this rule were not as spectacular as in release 1.

In release 3, the maximum number of defects per component found in development is seven. In system test it is 41. We set the threshold to four. Any component with more than four defects was considered fault-prone. A chi-square test on the data in Table 17 shows significant differences ($\chi^2 = 61.9964$; $p \leq 0.001$) between development fault-prone and development normal components in release 3.

In release 3, only one components was identified as fault-prone during development. The component was also fault-prone during system test. Applying guideline 1 would not have resulted in over-testing any components. There were three fault-prone components in system test, two of which were not fault-prone during development and would not have been tested enough according to guideline 1 (cf. Table 17). This latter misclassification is more dangerous.

Table 18 shows the results of applying the more conservative guideline 2. Chi-square test results are similar ($\chi^2 = 60.9803$; $p \leq 0.001$) when new components are grouped in with components that are fault-prone in development. While we “over-test” two more components than with the first guideline, we only undertest one (old) component that is fault-prone during system test.

Next, we evaluated whether system test defects are a good indicator of post-release defects for release 3. Table 19 shows the results. Chi-square analysis for data in release 3 shows no significant difference ($\chi^2 = 0.0328$; $p \leq 1$) between components that are fault-prone in system test versus those that were normal in predicting fault-proneness after release. Again, they point out that any component that was fault-

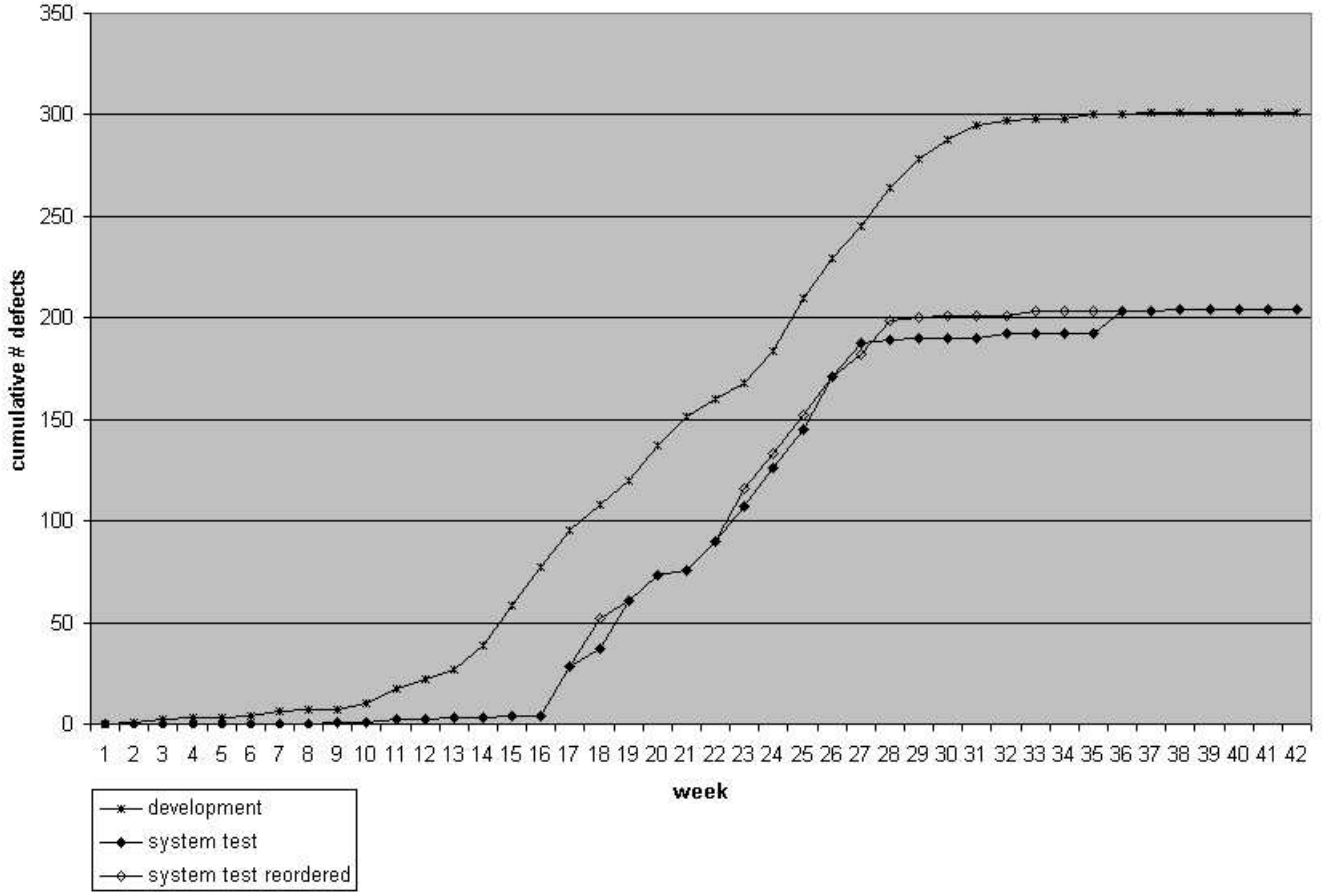


Figure 2: Cumulative defects for development, system test and reordered system test in release 2.

Threshold ≥ 4	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	1	0
Development Normal	2 (1 new)	185 (2 new)
Chi-square Results	$\chi^2 = 61.9964; p \leq 0.001$	

Table 17: Diffusion Matrix for Release 3.

prone during test was **not** fault-prone after release: The problems that were identified during testing were corrected before release.

Test guideline 3 specifies that components that were fault-prone during development should be tested as early as possible in a drop. As in release 1, we evaluated the effect of this guideline by shifting testing of development fault prone components to an earlier week in the same drop. Figure 3 shows the results. There were fewer defects in release 3 than in release 1, and thus the benefits of this rule were not as good as in release 1.

Threshold ≥ 4	Prediction (System Test)	
	Fault-prone	Normal
Development Fault-prone	2 (1 new)	2 (2 new)
Development Normal	1	183
Chi-square Results	$\chi^2 = 60.9803; p \leq 0.001$	

Table 18: Diffusion Matrix including new components in fault-prone category in Release 3.

Post Release Threshold ≥ 2 System Test Threshold ≥ 4	Prediction (Post Release)	
	Fault-prone	Normal
System Test Fault-prone	0	3 (2 new)
System Test Normal	2	183 (1 new)
Chi-square Results	$\chi^2 = 0.0328; p \leq 1$	

Table 19: Diffusion Matrix for System Test versus Post Release in Release 3.

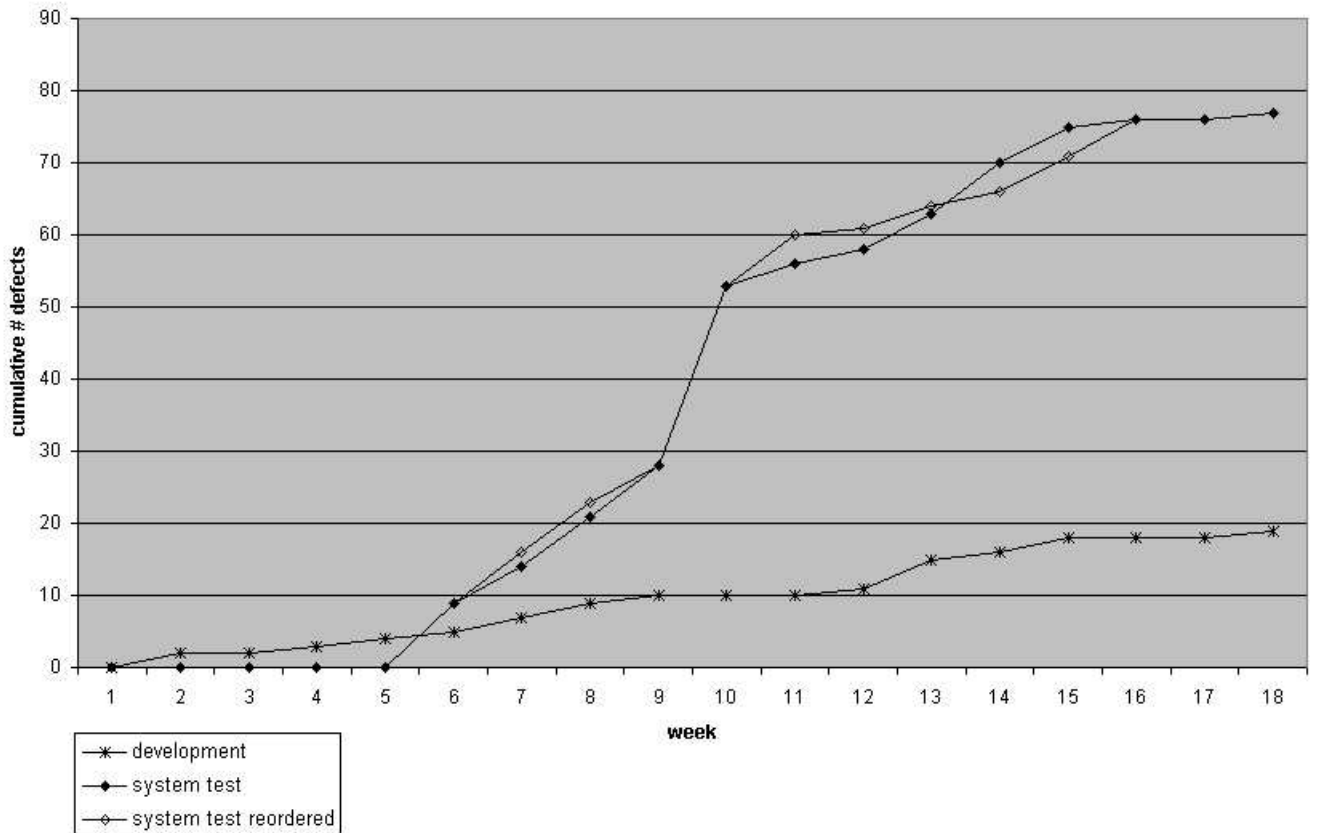


Figure 3: Cumulative defects for development, system test and reordered system test in release 3.

4.6 Cross release analysis

When successive releases are to be tested, it might be useful to have a test guideline on how to treat components that have been fault-prone in development, test, or in the field in prior releases.

A test guideline would be to consider a component fault-prone (and focus attention on it) in system test, if it was fault-prone during system test or in the field in the prior release. Tables 20 and 21 show the diffusion matrix analyses of this approach for the medical record system. Table 20 shows significant results applying this approach to releases 1 and 2 ($\chi^2 = 24.2612$; $p \leq 0.001$), while Table 21 shows that when the approach is applied to release 2 and 3, the results are not significant ($\chi^2 = 3.7056$; $p \leq 0.001$). The diffusion matrix results indicate that components that were fault-prone earlier, cannot be used to

	Release 2			
Release 1	Test Fault-prone	Test Normal	PostRelease Fault-prone	PostRelease Normal
Test or PostRelease Fault-prone	3	9	3	9
Test or PostRelease Normal	2 (2 new)	171 (3 new)	5	168 (5 new)
Chi-square Results	$\chi^2 = 24.2612$; $p \leq 0.001$		$\chi^2 = 13.2589$; $p \leq 0.001$	

Table 20: Diffusion Matrix for fault-prone components across Releases 1 and 2.

	Release 3			
Release 2	Test Fault-prone	Test Normal	PostRelease Fault-prone	PostRelease Normal
Test or PostRelease Fault-prone	1	11	2	10
Test or PostRelease Normal	2 (2 new)	174 (1 new)	0	176 (3 new)
Chi-square Results	$\chi^2 = 3.7056$; $p \leq 0.001$		$\chi^2 = 29.6488$; $p \leq 0.001$	

Table 21: Diffusion Matrix for fault-prone components across Releases 2 and 3.

predict fault-proneness in the next release. This is because problems, once identified, are fixed.

We also investigated whether there are some components that are not fault-prone during system test, but are fault-prone in development and after release. This would be an indication of insufficient testing. We found only one component that was fault-prone in development and post release in all three releases that was not fault prone in system test in any of the releases. This component needs to be tested more thoroughly. Further, there was only one component that was fault-prone in development, system test and post release for the first two releases and fault-prone in system test in Release 3. This is not so much an indication of insufficient testing, but of insufficient repair. In both cases, the number of such components is very small (one). This speaks for the quality of the development organization.

5 Conclusions

We evaluated development, system test and post release defect data to determine whether additional test guidelines based on this data might be helpful. The following guidelines were helpful.

1. When identifying fault-prone components, use an order of magnitude less than the maximum defects in a component to set the threshold for fault-prone components. This is less arbitrary than selecting a percentage of components as fault-prone.
2. Test components that are fault-prone during development more thoroughly. They are likely to be fault-prone during system test. De-emphasize testing of components that are not fault-prone during development.
3. Test new components more thoroughly, whether they are fault-prone during development or not.
4. Test components that are fault-prone during development as early in the drop as possible. It will give development more flexibility to fix defects and allow greater savings in test time when using statistical stopping rules for release decisions.
5. Between releases, pay particular attention to components that had development and post release problems, but where system testing found few problems (in our case, this was only one component). Likewise, evaluate the component that had problems in development, test, and after release in the prior release.
6. Improve impact analysis of enhancements to existing components and determine whether improvements in regression testing could have prevented problems from slipping through.

The following ideas are not helpful.

1. Analyzing defects found in components by severity level. We surmise that this is because severity is not a good indicator of how complete repair will be.
2. Setting the threshold for fault-prone components to percentage values, especially when the quality of the system is not known.

This being a case study, one cannot expect these guidelines to improve every project, although they certainly are sensible. The project studied has characteristics that need to be taken into account when determining whether applying these guidelines would improve system test performance:

- Few problems remain undetected.
- The number of post release problems is very low.
- Most known problems are fixed before release.
- In each release, the code is of very high quality. Only two components out of 188 are fault-prone in all releases.

One might say, we are “gilding the lily”. On the other hand, guideline 3 shows the potential of saving weeks of testing time and hundreds of thousands of dollars without penalty. Further, even the “almost perfect” project can benefit from the guidelines we derived from the defect data. For high quality development environments like the one analyzed, the key issue for testers is where to put emphasis, and where not to. This has the greatest potential for being more efficient without sacrificing effectiveness.

References

- [1] D. Ash, J. Alderete, P.W. Oman, B. Lowther, "Using Software Models to Track Code Health", *Procs. of the International Conference on Software Maintenance*, (September 1994), Victoria,, British Colombia, Canada, pp. 154 – 160.
- [2] S.Biyani, P. Santhanam, "Exploring Defect Data from Development and Customer Usage on Software Modules over Multiple Releases", *Procs. of the Ninth International Symposium on Software Reliability Engineering*, (November 1998), Paderborn, Germany, pp. 316 – 320.
- [3] S. Eick, C. Loader, M. Long, L. Votta, S. Vander Wiel, "Estimating Software Fault Content Before Coding", *Procs. of the 14th International Conference on Software Engineering*, (1992), Melbourne, Australia, pp. 59 – 65.
- [4] N. Fenton, N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System", *IEEE Transactions on Software Engineering*, vol. 26, no. 8 (August 2000), pp. 797 – 814.
- [5] P. Frankl, R. Hamlet, B. Littlewood, L. Strigini, "Evaluating Testing Methods by Delivered Reliability", *IEEE Transactions on Software Engineering*, vol. 24, no. 8 (August 1998), pp. 586 – 601.
- [6] T.M. Khoshgoftaar, R.M. Szabo, "Improving Code Churn Predictions During the System Test and Maintenance Phases", *Procs. of the International Conference on Software Maintenance*, (September 1994), Victoria, British Colombia, Canada, pp. 58 – 66.
- [7] T.M. Khoshgoftaar, E.B. Allen, "Predicting the Order of Fault-Prone Modules in Legacy Software", *Procs. of the Ninth International Symposium on Software Reliability Engineering*, (November 1998), Paderborn, Germany, pp. 344 – 353.
- [8] J. Musa, A. Iannino, K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, NY, 1987.
- [9] J. Musa, "Applying Failure Data to Guide Decisions", *Software Reliability Engineering*, McGraw-Hill, New York, NY, 1998.
- [10] N. Ohlsson, M. Helander, C. Wohlin, "Quality Improvement by Identification of Fault-prone Modules Using Software Design Metrics", *Procs. of the International Conference on Software Quality*, (1996), Ottawa, Canada, pp. 1 – 13.
- [11] M. Ohlsson, C. Wohlin, "Identification of Green, Yellow and Red Legacy Components", *Procs. International Conference on Software Maintenance*, (November 1998), Bethesda, Washington, D.C., pp. 6 – 15.
- [12] M. Ohlsson, A. von Mayrhauser, B. McGuire, C. Wohlin, "Code Decay Analysis of Legacy Software through Successive Releases", *Procs. of the IEEE Aerospace Conference*, (March 1999), Section 7.401.
- [13] M. Sahinoglu, A.K. Alkhalidi, (1997) "A Compound Poisson LSD Stopping Rule for Software Reliability", *Procs. of the 5th World Meeting of ISBA, Satellite Meeting to ISI-97*, (August 1997), Istanbul.
- [14] N.F. Schneidewind, "Software Metrics Model for Quality Control", *Procs. of the International Symposium of Software Metrics*, (November 1997), Albuquerque, New Mexico, pp. 127 – 136.
- [15] A. von Mayrhauser, J. Wang, M. Ohlsson, C. Wohlin, "Deriving a Fault Architecture from Defect History", *Procs. of the International Symposium on Software Reliability Engineering*, (November 1999), Boca Raton, Florida, pp. 129 – 146.

- [16] C. Wohlin, P. Runeson, “Defect Content Estimations from Review Data”, *Procs. of the International Conference on Software Engineering*, (April 1998), Kyoto, Japan, pp. 400 – 409.
- [17] C. Wohlin, P. Runeson, “An Experimental Evaluation of Capture-Recapture in Software Inspections”, *Journal of Software Testing, Verification and Reliability* vol. 5, no. 4 (1995), pp. 213 – 232.