

Test and Development Process Retrospective – a Case Study using ODC Triggers

Ram Chillarege & Kothanda Ram Prasad
CHILLAREGE INC., NEW YORK, APRIL 2002
www.chillarege.com

Abstract

We present a case study of a product development retrospective analysis conducted to gain an understanding of the test and development process effectiveness. Orthogonal Defect Classification (ODC) is used as an analysis method to gain insight beyond what classical qualitative analysis would yield for the probable cause of delays during test.

- 1. ODC Trigger analysis provides the insight to understand the degree of blockage in test, probable cause, and consequences to the test and development process.*
- 2. Trigger distribution changes with respect to time shows the stabilization of the product, and variation among components shows the systemic nature of issues.*
- 3. The study makes nine specific inferences and recommendations based on these analyses to guide the engineering of future releases*

Introduction

At the conclusion of a development effort, it is quite common for a project to conduct a postmortem, or what is also called a retrospective. The value of such an exercise is to gain an understanding of the process and learn from mistakes. There are no set methods on how to conduct such retrospectives, but the process and spirit of such an exercise is quite well understood. However, the degree of sophistication used for such analysis varies widely, and consequently the degree of insight and understanding gained.

Informal analysis is by far the most commonly used approach. The opinions and observations of the team are collected and consolidated to generate conclusions. Often, it becomes hard to resolve conflicting views or establish diagnostics for clear differentiation of causes. By and large, the role of quantitative methods to support such retrospective analysis is limited to the few organizations that have the necessary resources and skills in software engineering process analysis.

Orthogonal Defect Classification (ODC) is a measurement system for software processes based on the semantic information contained in the defect stream [Chillarege 92]. It brings a new level of sophistication to such analysis and leverages information captured in the development process. Today it provides objective methods to gain clarity where in the past one relied purely on intuition and experience [Bassin 98]. ODC methods have been expanded to multidisciplinary design as reported in [Amezquita 96]. Recent case studies at IBM are reported in [Butcher 02] and at Cisco in [Mullen 02].

A common misconception is that the full scope of ODC is needed for such analysis. While the full scope of the data is most certainly valuable, it is not always necessary. The scope of data needed is dependent on the types of questions raised. Often, even a restricted set of data can prove valuable.

This case study uses ODC Triggers to analyze the efficiency of the software test subprocess. Trigger is our key to gaining an understanding of what happened during a test cycle and its efficacy.

Additional data such as ODC defect type and ODC source [Chillarege 92][ODC Web] would be useful to further narrow down to the specific development practice problems, but are not critical for the first round of analysis.

ODC triggers can be easily generated from the defect tracking logs. For retrospective analysis, triggers tend to be easy to extract from test defect logs, whereas some of the other ODC attributes may not be as easy. This works in our favor for this exercise, since much of the analysis needs triggers - probably the most valuable data when studying the test subprocess.

While this paper is a case study in retrospective analysis, it also brings together ideas to deliver a more complete picture. Several earlier papers discussed specific technical issues. We build on them, exploiting measurement to provide insight far beyond what classical qualitative retrospective analysis would do. From an academic perspective, we are helping develop the area of experimental research in computer science [Iyer 90].

The rest of the paper is organized thus. We begin with a brief case history and set the objectives for this detailed retrospective. The data and analysis section that follows has several subsections of analysis. The defect profile and the defect priority capture quantitatively what was experienced qualitatively. This is followed by detailed trigger analysis. A brief description of triggers is included to help the reader who is not too familiar with ODC. Trigger distributions are studied for the entire test cycle, and as a function of time and component. These analyses are used to summarize our findings, under a section called inferences and recommendations. Nine specific findings are summarized that capture the insight and diagnosis for the delays experienced in test. The findings also translate to recommendations for engineering improvement in the next release. The conclusions circle back to address questions raised in the objectives section.

Case History

The project being developed is the fourth or fifth iteration of an enterprise application: a typical modern three tier web application, with databases, access control, and zero footprint clients. The backend is a relational database that contains the enterprise data and is also a container for some of the business rules. The application layer is supported by an application server and business objects. There is an authentication and security component, report generation subsystems, business rules management, import and export of data, and a rendering layer to handle the dynamic data. Since these applications are replacing many of the classical client-server applications, some of which still run on IBM 3270 screens, there is a considerable amount of work to blend the usability of the modern web face of the application to the efficiency, speed and simplicity of the classical mainframe applications. This is a difficult task, since the designs needed to emulate the speed and simplicity of UI in mainframe applications yet compete with the desktop style, while not compromising too much performance for similar functionality.

The major components are fairly standard for such application and include the backend relational database, authentication, business logic, customization of the logic and workflow, data import and export, views and reports and dynamic rendering engines, etc.

The logic of the application is quite well understood since this is the n^{th} iteration with essentially the same core business logic. The business growth had put demands on performance and usability and also added to the number of business rules. Such growth usually placed demands in all dimensions of the product - it is not merely a feature increase, but one that needs to accomplish feature growth with overall improvement in quality and appeal.

Level 2 Key Process Areas (KPA)	
Requirements Management	Requirements documented for each module and kept up-to-date.
Project Planning Project Tracking and Oversight	Function Points used for estimation. Weekly reporting and Defect tracking
Configuration Management	SCM for documents, code and test cases
Software Quality Assurance	No formal SQA in place
Subcontract	Not Applicable
Level 3 Key Process Areas (KPA)	
Training, Intergroup coordination & Peer Reviews	Existed
Integrated software development	Partially met
Organizational Process focus, Definition, and software product engineering	Minimally met

Table 1 Our “quick and dirty” assessment places the project’s SEI CMM software maturity between a Level 2 and Level 3.

There has been greater maturity in the development team through evolution. While formal assessments of Software Engineering Institute’s Capability Maturity Model, (SEI CMM) [Humphrey 89] [Paulk 93] were not conducted, we provide a rough assessment shown in Table 1. Based on the KPAs, we place the team at a maturity level somewhere between a Level 2 and a Level 3.

Earlier experience with this application had taught the development team the value of careful requirements management. A focus had been placed

on understanding customer requirements and reviewing them with prospective clients. The Quality Assurance (QA) team had also been involved in reviewing such requirements and developing test cases from them.

Given the degree of change in the new release, a thorough design review was necessary to alter the architecture and performance of the application. Changes were separately prototyped and evaluated with test data so that the architecture could be validated before migrating the code. This provided substantial isolation of the core of the application from the business rules and user interface. The techniques for writing business rules were also prototyped and tested so that development could begin without too many unknowns in the data and control flow through the application.

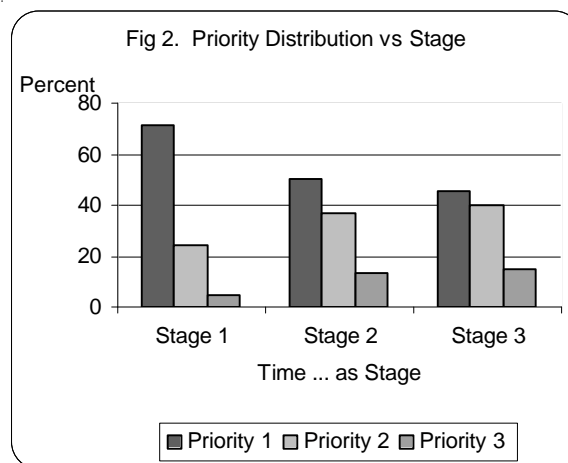
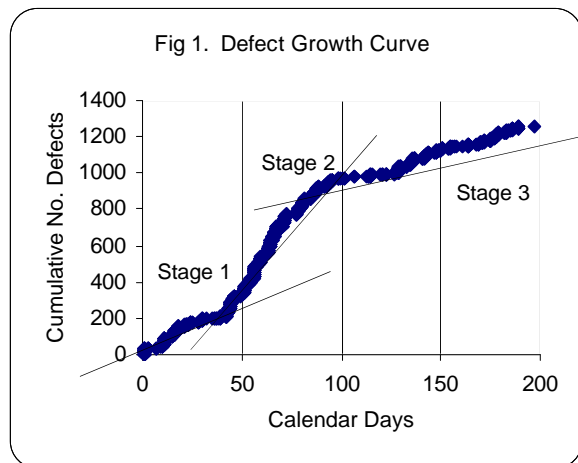
Testing the new business rules required that most of the application worked. Given the degree of interaction with the various components, it was not possible to isolate business rules and test them on a test harness. Therefore the QA team diligently developed a test suite that covered the entire set of requirements. A regression bucket was built, such that an automated test execution platform could run against the business rules.

Objectives

The primary objective in this detailed quantitative retrospective is to gain a clearer understanding of the development and test processes. There was a belief that test did not perform as well as it could have, and was slow. Is this true? And if so, why?

There were several process improvements engineered into this development process, as discussed in the case history. There was considerable focus on architecture and design, a very thorough requirements process, and a well-articulated development and build mechanism.

Yet, the experience through test was prolonged and it was felt that the progress was slower than expected. In spite of that, the product stabilized and customers seemed satisfied.



Figures 1 and 2 illustrate quantitatively some of the challenges experienced qualitatively. The high proportion of priority 1 defects blocked the progress of test - another way to communicate the sense that test was slow. The stages 1, 2, 3 divide the time line and S curve into distinct phases of defect detection. Note that while there is a pronounced shift in the priority distribution among the stages, the fraction of priority 1 and 2 defects is still quite large. The reasons behind this, or the “why?”, needs ODC trigger analysis to gain insight, corroborate the evidence, and shed light on possible solutions.

Some of the questions we ask are:

- Was the test process independently responsible for the delays?
- Were there elements of development that could have helped the test cycle?
- Was there a visible benefit gained from the good design and architecture process?
- Was there a visible benefit gained from the more thorough requirements process?
- What would make the test process more effective?

The answers to these questions need to be substantiated with detail rather than a mere yes, or no. The data and analysis section investigates probable cause to summarize the inferences and recommendations. In conclusion we tie back the detailed findings to our higher level objectives.

Data & Analysis

Defect Profile

The formal test cycle was overlapped with the development process, as is common in many a

development process. It began after a substantial amount of the code was available for testing. This happened when at least 50% of the modules were ready and the application could be successfully built with features available for testing. New code is added to the base in regular intervals and the build schedule is known. Thus the test function is aware of the new functionality included in each build.

Figure 1 shows the cumulative number of defects drawn against calendar days. To understand the delays in testing we have divided the time line into three stages, each of which demonstrates a significant difference from the other in defect detection rates.

- Stage 1: Day 0 - Day 40
- Stage 2: Day 40 - Day 100
- Stage 3: Day 100 - Day 200

The periods themselves do not have any significance from a development process or mark an event - they are merely sections of the S curve. Stage 1 displays a very slow progress in defect detection, while stage 2 experiences a more accelerated defect detection. Stage 3 is the remainder of the test cycle where the defect detection rate further slows down.

Defect Priority

The Priority of a defect indicates the urgency expressed by the test organization for a fix. It does not have as much to do with the severity of the failure as it does to the immediacy of fix to continue with testing. Thus, Priority and its distribution tells us about the current state of affairs in test and its consequences on the test cycle.

Severity, a popular measure of the impact of a fault, sounds very similar to priority. We want to make clear that, in this paper, we use priority and not severity. Severity, becomes quite useful towards the end of the development life cycle, while its use in early parts of the function test is debatable.

Priority 1 defects imply that additional testing cannot continue until the fix for the failure is provided. Priority 2 defects indicate a problem that also reduces the effectiveness of test but is not a critical show stopper. Priority 3 defects are of a lesser significance; in that they need to be fixed, however, they do not hinder the progress of test.

A larger fraction of priority 1 defects indicate that the testing process is not running smoothly. The bugs are probably blocking bugs, and the effectiveness of the test organization is limited. There is consequently much back and forth between development and test, which impacts the productivity of both organizations.

Figure 2 shows us the distribution of defect priority and how that changes from stage 1 to stage 3 of this test cycle. The fact that during stage 1 the percent of priority 1 defects was as high as 70%, explains why the progress of test was slow. During stage 2 the percent of priority 1 drops by 20% and the rate at which defects are being detected rapidly increases. Finally, during stage 3 when the product was stabilizing the percent of Priority 1 defects had dropped to its lowest in this test cycle.

These data, illustrate the magnitude of the hurdles faced during test. The large fraction of priority 1 defects reinforces the sense that was communicated by management that the test cycle was prolonged. It does not immediately offer an explanation of why this is the case, but makes clear that the problem is real. Further insight into the cause of this would be difficult to establish without using ODC Triggers.

ODC Trigger

Software triggers are the catalysts which allow the defect to surface. Every time a review or test is done, that activity is exercising a particular trigger which may allow a defect to surface. While a test case examines functionality or checks a requirement, the way it does so is captured by the trigger. The different triggers express the range of ways by which tests surface bugs.

Software triggers are discussed at length in the original ODC work; the collection of papers available on-line at [ODC Web]. A good discussion on what triggers are, and the properties they possess can be found in [Chillarege 95]. Table 2 lists software triggers and an association with process phases that typically generates them.

Since ODC attribute values are carefully named they are usually descriptive enough that one can follow the discussion on the data with ease. However, it certainly helps to have the definitions handy. For the purposes of this paper we have provided a short two line definition of the relevant triggers in the following subsection.

Short descriptions of relevant Triggers

DESIGN CONFORMANCE: A review trigger where the implemented design is compared against a reference - either design document, pattern, or guideline.

LOGIC/FLOW: Checking for correctness or flaws using knowledge of the practice.

BACKWARD COMPATIBILITY: Examining compatibility with prior version of the product.

LATERAL COMPATIBILITY: Examining for compatibility with other products and platforms that need to work with this release.

CONCURRENCY: Serialization, shared resources, multi-threaded tasks, timing, etc.

INTERNAL DOCUMENT: Inconsistencies in prologs, and sections in the same work product.

LANGUAGE DEPENDENCY: Programming standards, specific implementation considerations, environment restrictions, execution modes, etc.

SIDE EFFECTS: Usage behavior beyond design, but relevant in context. Do A; B happens.

TRIGGER	D	C	U	F	S
Design Conformance	X	X			
Logic/Flow	X	X			
Backward Compatibility	X	X			
Lateral Compatibility	X	X			
Concurrency	X	X			
Internal Document	X	X			
Language Dependency	X	X			
Side Effects	X	X			
Rare Situation	X	X			
Simple Path			X		
Complex Path			X		
Coverage				X	
Variation				X	
Sequencing				X	
Interaction				X	
Workload Stress					X
Recovery					X
Startup/Restart					X
Hardware Configuration					X
Software Configuration					X
Blocked Test/Normal Mode					X

KEY: D: Design Review, C: Code Review
U, F, S: Unit, Function, System Test

Table 2. Triggers are commonly associated with a verification stage that is likely to generate the particular trigger. Thus, in ODC vernacular one could refer to a trigger as, “ a review trigger”, implying that the trigger belongs to the set of triggers usually encountered in activities of review. We must note that not all triggers associated with a stage are necessarily experienced in any specific implementation.

RARE SITUATION: Unusual issues related to idiosyncrasy of environment, hardware, or software.

SIMPLE PATH: White box - Straightforward usage of code path and branches.

COMPLEX PATH: White box - Exercising conditionals, and circuitous coverage.

COVERAGE: Black box - Straightforward usage of function or single parametrized execution.

VARIATION: Black box - straightforward like coverage, but with a variety of parameters.

SEQUENCING: Black box - multiple functions, with a specific sequence.

INTERACTION: Black box - when two or more bodies of code are involved.

WORKLOAD STRESS: Pushing the limits of performance, resources, users, queues, traffic, etc.

Recovery: Invoke exception handling, recovery, termination, error percolation, etc.

STARTUP/RESTART: Major events of turning on, off or changing the degree of service availability.

HARDWARE CONFIGURATION: Issues surfaced as a consequence of changes in hardware setup.

SOFTWARE CONFIGURATION: Issues surfaced as a consequence of changes to software setup.

BLOCKED TEST/NORMAL MODE: Test could not run during system test. Normal Mode - *archaic*: customer found nonspecific trigger.

The overall Trigger Distribution

We start by looking at the overall trigger distribution. The Figure 3 shows the triggers of all defects over the entire test cycle, a population of around a thousand defects. The purpose of this distribution is to quickly assess the range of triggers present in the entire test. We expect the process has generated a wide range of triggers, or at least those that can be expected from a function test and some of the system test triggers. We will later look at them as a function of time, to see if the changes are along what ODC would expect from an evolutionary and stabilization perspective.

It is evident, from the figure, that the coverage and variation triggers dominate the entire distribution. That there are very few other triggers in the distribution, and that Coverage triggers dominate more than 50% of all the triggers are indicative of a potential problem - either in development or test. While Coverage normally does tend to be one of the largest triggers in function test, when it is not supported by the other triggers it raises suspicion.

A coverage-triggered defect could also have been found through unit test or inspection. In fact, inspection is a particularly good mechanism to find coverage triggered defects of function test because they mostly have to do with ensuring logic flow and completeness of function. While automated testing is also good at finding these defects, the burden that

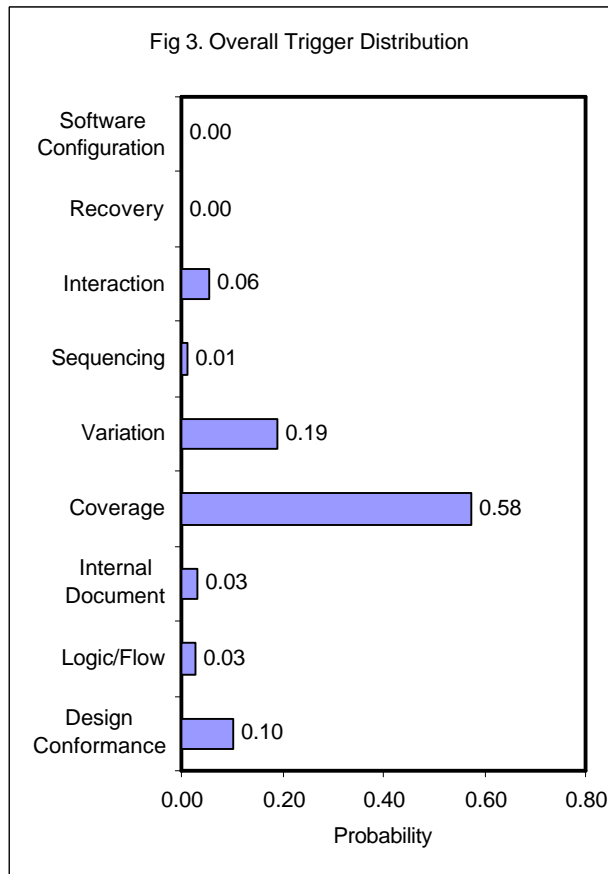


Fig 3. The overall trigger distribution is dominated by coverage triggers - the type of trigger that can also be found by other means such as inspection and unit test. NB. the population is over a 1000 defects.

is generated is because of the blockage that can be caused as a consequence of these faults. By blockage, we mean that if a basic functionality fails, then a series of functionalities that are dependent on the failed functionality also fail or those functionalities cannot be tested. e.g. If creation of an object fails, then all other test cases depending on the object cannot be executed.

This product had around 1000 function points, and since the total number of defects is approximately 1000, the test found defect rate is about 1 defect per function point. Whereas, the industry average for this class of code is around 0.5 defects per function point [Jones 98].

It is not uncommon for new code from a development team that is under a considerable pressure to have an increased defect injection rate. A factor of two is not too bad given that the variance in defect injection rates is wider than that. The critical issues are really the nature of defects and the efficacy of the test cycle. The issue becomes one of the length of time it takes to test and the nature of testing that is required to clean up the code.

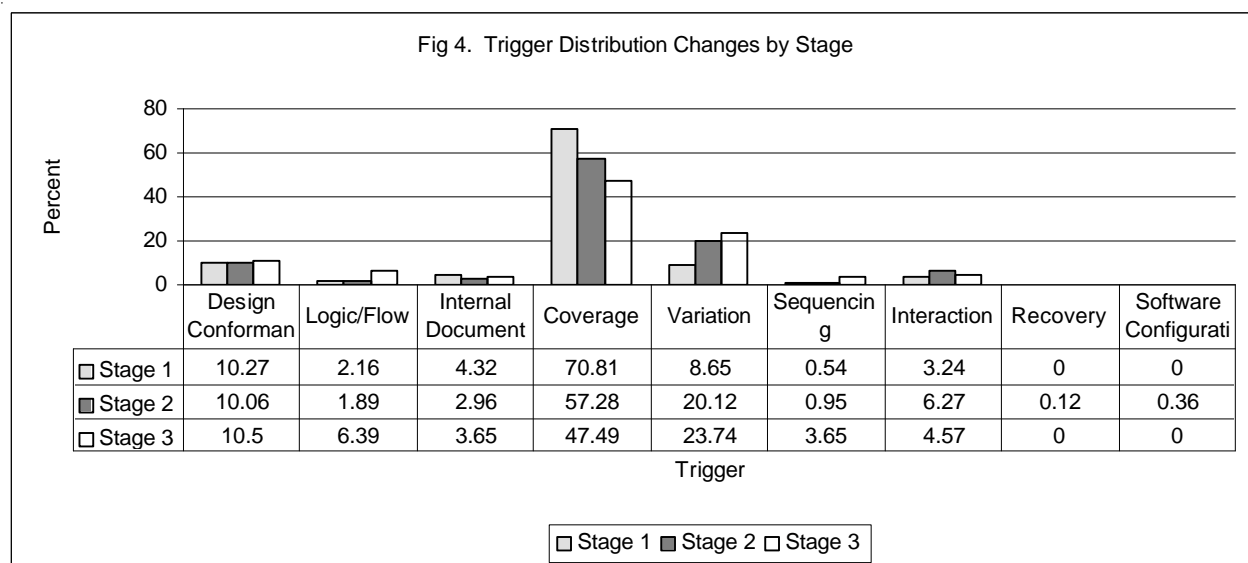
That these defects are coverage dominated and that the overall defect rate was large is a clear indication that the quality of code that came through from the development phases was poorer than it could be. Many of these defects are best found through inspection, and that process was definitely weak.

This burden of high coverage triggered defects at twice the acceptable volume now explains why during stage 1 and stage 2 the test progress was burdened. (Correspondingly, the probability of a priority 1 defect was between 0.5 to 0.7, which is high and told us of the high burden and block).

Clearly, an opportunity to reduce the test cycle time would be to address defects of this trigger before the code enters test. It also becomes evident that the return on investment for such process action is huge. The issues are one of training and targeted inspection lists, as opposed to exhaustive lists. The ODC Triggers, taken together with the ODC Type and ODC Source can help generate such a list. This is an ODC Action step, a process discussion that is beyond the scope of this paper.

Trigger vs. Time

Figure 4 shows the defect trigger distribution as it changes from stage 1 to stage 3. In each of these stages, the overall nature of the trigger distribution is very similar to that of the aggregate distribution. Namely, coverage dominates the distribution. However, what is interesting to note is that the proportion of coverage defects also reduces from stage 1 to stage 2 and then further in stage 3. The decrease in coverage is picked up by an increase in variation. The more complex triggers such as sequencing and interaction, continue to have a small profile.



Component:	A	B	C	D	E
No. of Defects	189	300	358	135	114
Trigger % for component					
Design Conf.	9	11	10	13	10
Internal Doc.	4	3	5	0	3
Coverage	66	51	56	58	68
Variation	12	24	19	16	10
Sequencing	2	0	2	3	1
Interaction	3	6	6	7	8
Recovery	0	0	0	0	0
Soft. Config.	1	0	0	0	1
Lat. Comp.	4	4	2	3	0

Table 3. Trigger proportions by component

Figure 4 and Table 3 show the distribution of Triggers as a function of time (stage 1 to 3) and as a function of component. What is quite evident is that there are no significant variation in either dimension. The large coverage and variation triggers are present both by stage, and by component. However, there is a subtle but distinct shift in distribution as test progresses from stage 1 to stage 3. This is evidence of the stabilization, albeit not as pronounced as we would have liked to see.

This change in the trigger distribution, where the fraction of coverage triggers is dropping is evidence of the stabilization of the product. Software reliability growth models use defect rate curves to detect such stabilization [Musa 99]. While the rate curves can illustrate growth, they are not as clear as to what can be visualized in the change of the trigger distribution. Further, measuring growth is hard during the function test phase. It is a more viable method during system test, and early customer trials. Function test is very dependent on test plan execution issues – such as those related to build and integration

methods. Triggers, however, are not subject to those dynamics and provide a finer level of information. But, do note that the trigger data does not preclude defect rate analysis, which can be profitably combined as discussed in [Chillarege 94].

Trigger vs. Component

To further investigate the trigger distributions, we look at the trigger distribution by each of the different components. Table 3 shows cross-tabs of the triggers versus the major components. We did not include

the chi-square test since the sparseness of the table yields over 20% of the cells with counts less than 5. But the key conclusions are quite apparent. Coverage and Variation triggers dominate the distribution in all components. Even the proportions are very similar, indicating that the discussion in this paper is equally applicable to all of the product's subcomponents. The problems and issues are systemic in nature, which was suspected, but is now indicated. The inferences drawn in the following section apply uniformly to the entire product and are representative of the development process and culture.

Summary: Inferences & Recommendations

1. The test process was completely dominated by coverage triggers [Fig. 3]. This is a systemic problem persistent in all components of the product. While it is not unusual for coverage triggers to dominate test at a functional test level, this particular proportion was much larger than normal.
2. The large fraction of coverage-triggered defects indicates a weakness in code inspection and/or unit test [Tbl. 2, ODC refs]. We had suspected weakness in the code inspection stages, but the magnitude and consequence is made visible through trigger analysis.
3. The high coverage triggered defects are also associated with the high priority 1 defects. This explains the slowness of the testing progress in stage 1. Although, there could be other factors such as code delivery and build issues, the disproportionately large coverage (~ 60%) singly explains the delays in test.
4. The change in trigger mix as we progress from stage 1 to 3 is the most compelling evidence of the stabilization of the product [Fig. 4]. Growth curves could not so readily be applicable here given the type of data and measurement that we are dealing with in a largely function test environment.
5. It must be noted that sequencing and interaction triggers alone do not indicate complete stabilization [Fig. 4, Tbl. 3]. One would like to see more workload and stress related triggers, which were not present. This indicates that while some of the latter tests were probably of a systems test nature, there could be a fallout of those faults during early field usage.
6. The rather few logic trigger errors and few design conformance triggers are a very positive sign [Fig. 3]. Clearly, the effort invested by the team toward more thorough design reviews and modeling has paid off. Thus, these functional tests have uncovered rather few design, architecture and requirement related defects.
7. The defect injection rate which is approximately one defect per function point is about twice the industry average [Jones 98]. This is not a bad sign, largely because of the nature of the bugs. These data tell us that there can be a substantial savings in test cost for the next release if the learning from this retrospective can be gainfully applied.
8. Actions are needed to improve the module level design and code stages. Better inspection and unit test can cut down a significant size of this defect volume at lower cost [Humphrey 89]. Additionally, a drill down using other ODC attributes can further localize the check lists.
9. Reducing the test cycle time by a factor of two is not unthinkable. All things considered, targeted inspection can cut this defect escape rate by more than half, and the resources better used to gain reliability. There are several discussion on managing the test cycle reported in the literature. For example, a good treatment on the techniques of modeling and managing defect removal is discussed in [Levendel 91]. Our focus has been on identifying probable cause - which, once identified, opens up several possibilities for solutions and remedies. The discussion of different remedy strategies and their tradeoffs deserves a separate discussion, and can follow this study.

Conclusions

The objectives of this study laid out a few broad questions that can now be answered with significant clarity. The nine inferences and recommendations examine and argue the details to explain the “why” of the test experience. Now, we address the higher level questions posed in the objectives section.

The test process was clearly not independently responsible for the delays [Summary 1-4]. The analysis revealed the specific elements in the development process which could have improved the test cycle [Summary 7, 8]. However, this does not cast development in a bad light at all – there were clear benefits gained from the focus on the design and requirement process [Summary 6], and the magnitude of the problem is not alarming [Summary 7]. With this clearer understanding of what happened there are several methods possible to manage this process, deliver a shorter test cycle and achieve higher field reliability. While a detailed discussion on the design of such methods and processes is way beyond the scope of this short article, some have been mentioned in passing and a few references cited to related work [Summary 1-3, 9].

The case study illustrates the power of ODC analysis to gain a deeper understanding while unearthing possibilities for solutions and guidance for specific action. This is the fodder to drive very significant return-on-investment in software engineering.

Acknowledgments

We thank our clients, the referees for their critique, and Saroja Prasad and Sunita Chillarege for painstakingly editing the paper.

References

- [Amezquita 96] “Orthogonal Defect Classification Applied to a Multidisciplinary Design”, A. Amezquita & D.P. Siewiorek, CMU EDRC 05-100-96.
- [Bassin 98] “Evaluating Software Development Objectively”, K. Bassin, T. Kratschmer, P. Santhanam, IEEE Software, Vol 15, 1998.
- [Butcher 02], “Improving software testing via ODC: Three case studies”, M. Butcher, H. Munro, T. Kratschmer, IBM Systems Journal, Vol 41, No. 1, 2002.
- [Chillarege 92] “Orthogonal Defect Classification - A Concept for In-Process Measurements”, Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, Man-Yuen Wong, IEEE Transactions on Software Engineering, Vol. 18, No. 11, Nov 1992.
- [Chillarege 94] “Identifying Risk using ODC Based Growth Models”, R. Chillarege, S. Biyani, Proceedings, 5th International Symposium on Software Reliability Engineering, IEEE, Monterey, California, pp 282-288, November 1994.
- [Chillarege 95] “Software Triggers as a function of time - ODC on field faults”, Ram Chillarege and Kathryn A. Bassin, DCCA-5: Fifth IFIP Working Conference on Dependable Computing for Critical Applications, Sept 1995.
- [Humphrey 89] “Managing the Software Process”, Watts S. Humphrey, Addison-Wesley 1989.
- [Iyer 90] “Introduction Experimental Computer Science”, R. K. Iyer, IEEE Transactions on Software Engineering, Vol 16, No 2., 1990.
- [Jones 98] “Estimating Software Costs”, T. Capers Jones, McGraw-Hill, 1998.
- [Levendel 91] “Reliability Analysis of Large Software Systems: Defect Data Modeling”, IEEE Transactions on Software Engineering, Vol 16, No. 2, 1990.
- [Mullen 02] “Orthogonal Defect Classification at Cisco”, R. Mullen, D. Hsiao, Proceedings ASM Conference, 2002.
- [Musa 99] “Software Reliability Engineering”, McGraw-Hill, 1999.
- [ODC Web] Orthogonal Defect Classification
www.chillarege.com/odc
www.research.ibm.com/softeng
- [Paulk 93] “Capability Maturity Model for Software, Version 1.1, Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, Charles V. Weber, Software Engineering Institute, 1993.