

2008

PACIFIC NW SOFTWARE
QUALITY
CONFERENCE



COLLABORATIVE
QUALITY

OCTOBER 13-15, 2008

CONFERENCE PAPER EXCERPT
FROM THE

CONFERENCE
PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

Dealing With the Most Influential Factors that Cause Customer Dissatisfaction, An Organization-wide Effort in Product Crash Reduction/Elimination

Abstract

Product crash is an abnormal termination of a software program, which is usually caused by a software issue, although a hardware failure can also be the reason. When a computer "crashes," it locks up (freezes), and the user cannot obtain any response from the keyboard or mouse. Product crashes represent a relatively small percentage of problems compared to the total number of product issues (e.g., ~ 20%). However, they have a disproportionate influence on product users. When crashes occur, they are very annoying to users as they often result in the loss of data and require immediate resolution before normal operations can resume.

This paper describes the story of how an organization addressed their product crash issues. It will discuss the success factors and best practices that were used in the context of the following sequence of key activities:

- Executive commitment and leadership
- Core team and organization involvement in the implementation
- Identifying error-prone products with Failure Analysis and Root Cause Analysis
- The metrics used to verify improvement
- The progress tracking and corrective action process
- The product crash reduction process
- The results
- Lessons learned and future improvements

The first year results confirmed the effectiveness of the effort in reducing the total number of crashes in the organization's products. The organization strongly believes that continuous process improvement and consistent implementation of the product crash reduction process will enable the organization to achieve its improvement objectives in customer satisfaction and engineering productivity by minimizing the cost of rework.

The experiences and lessons learned from this story can be useful for other companies with similar needs in customer satisfaction as well as product and process quality improvement.

Bio

Dr. Duvan Luong is an Engineer Director/Architect at Cadence Design Systems. Dr. Luong's background is in the software development process and best practices with emphasis in testing. Dr. Luong's other interests are in the areas of organizational change, process improvement, customer satisfaction, effective consulting and problems solving, and operational excellence including metrics and statistical controls. Dr. Luong has had 29 years of experience in the Software Development Industry with AT&T and Bell Labs, IBM, Synopsys, Sun Microsystems, Hewlett Packard, and Cadence Design Systems.

Introduction

For most successful software companies, effectively addressing the key factors of customer dissatisfaction and the high cost of rework are the top must-have items in the company's business strategy. It is widely known that product defects have a very high correlation with the above two factors. If a company can effectively handle its product defects, it has addressed the major contributors to both its customer satisfaction and the cost of rework issues.

As customers use their purchased software products they will likely experience many product "issues." Depending on the type of issue they encounter, the customer will have a varying amount of negative experience, and hence their satisfaction for the product is adversely affected. Crashes are one type of serious product issue. One definition of a "crash" is an abnormal termination of a computer program, usually caused by a software issue, causing a computer to lock up or freeze and preventing it from responding to other parts of the system such as the keyboard or mouse. While these are a relatively small percentage of the total issues (e.g., ~ 20%) they have a disproportionate influence on customer satisfaction. When crashes occur, they are very annoying to users as they often result in the loss of data and require immediate resolution before normal operations can resume. Crashes also result in wasted effort required for re-starting customer operations and for rework by product developers to implement fixes to the crashes.

The Story

This is a story of a software development organization of several hundred members who reduced the number of current and potential crash issues in their product code base of more than 15 million lines of source code. The organization achieved its crashes reduction objectives several years ago and is currently continuing to expand their improvement efforts to include other areas of product reliability.

When the product team had a meeting with a key customer some time ago to discuss product quality issues, to the team's surprise, the customer pulled-out records showing how frequently the product crashed in their operational environment. By the end of the meeting the message was very clear – "The company tools crashed too often. When they crashed, the crashes severely impacted customer operations. The customer was not very happy with this situation". The team that developed the product believed it could do many things to improve the customer experience with respect to crashes... The team then embarked on a journey to eliminate crashes.

As the first year of the improvement program came to an end, the team was starting to see some initial results from their improvement actions: 100% Unit Testing was achieved for the latest release for the first time (previously Unit Testing was not emphasized as part of the standard development practice). All product code was analyzed with a Static Analysis tool that identified many key issues that were promptly fixed. Root-cause analysis results started to show patterns of common issues so the team could formulate lasting effective resolutions for them. Total unresolved crashes came down by 70%. Best of all, customers started to have a better operational-reliability experience with the team's products.

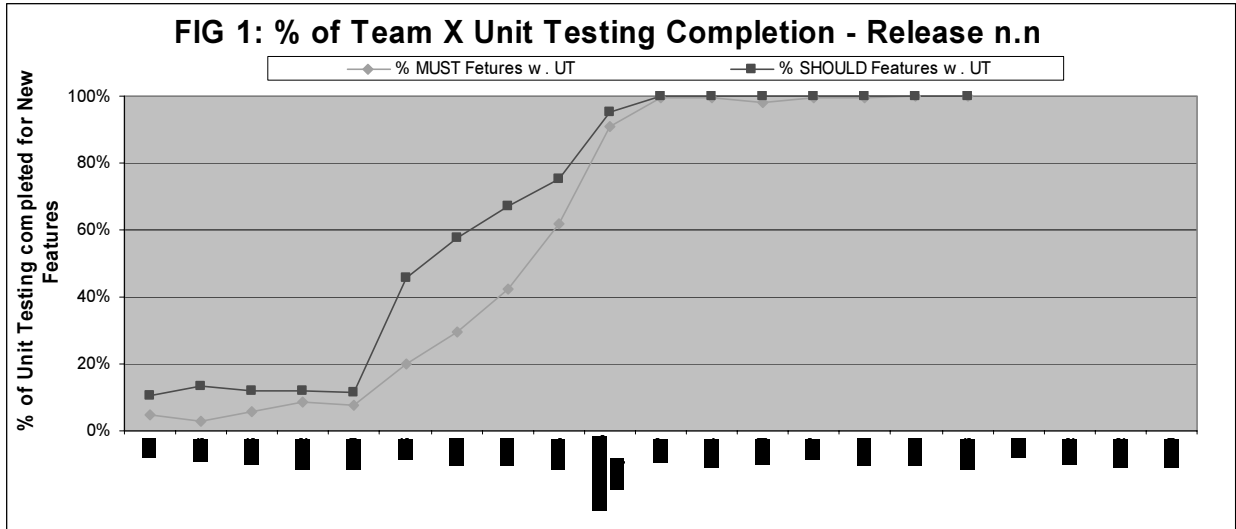


FIG1: % of Team X Unit Testing Completion - Release n.n

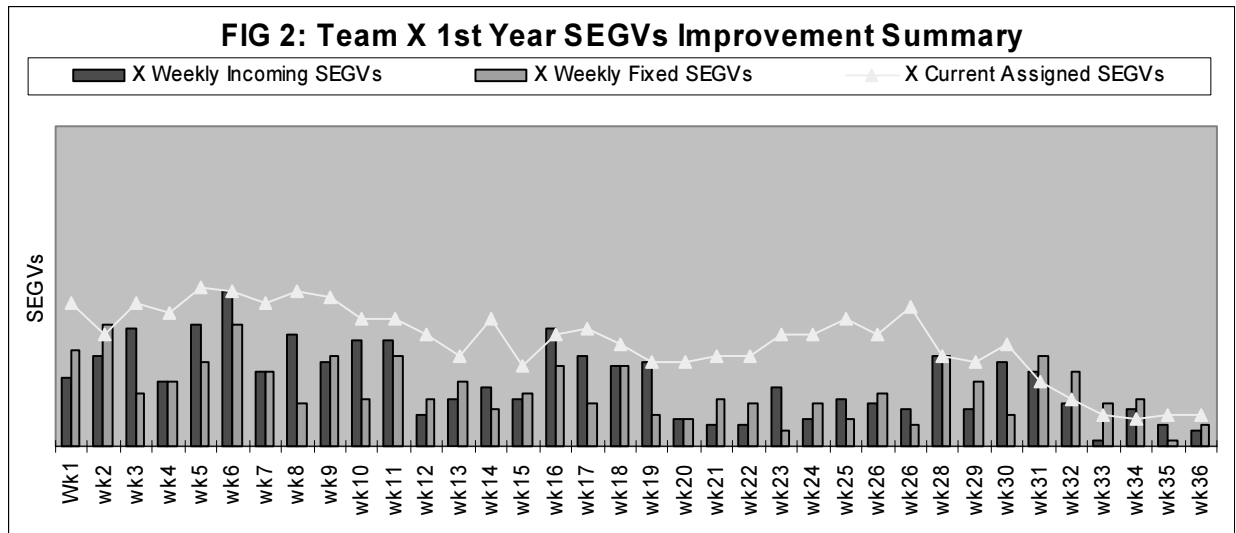


FIG 2: Team X 1st Year SEGVs (Crashes) Improvement Summary

As the team continued to make progress over the next several years the effort was shifted from crash reduction to resolving other kinds of product issues, moving from defect detection to prevention and from working on prior unresolved issues to managing new incoming issues. The team started the improvement effort with the belief that “Doing the same thing and hoping for different and better results is unrealistic.” The team now believes they have a new motto to share: “Consistent focus on making improvements will produce better results.” At the time this story started this particular software team was ranked at an undesirable position compared with other teams in the company in terms of product quality; however, the team gradually improved its position. As of today, it has already surpassed the aggressive quality improvement targets set by the company and is on the way to make even more improvements with its strong focus and momentum. The team has become the quality benchmark for many other teams in the company.

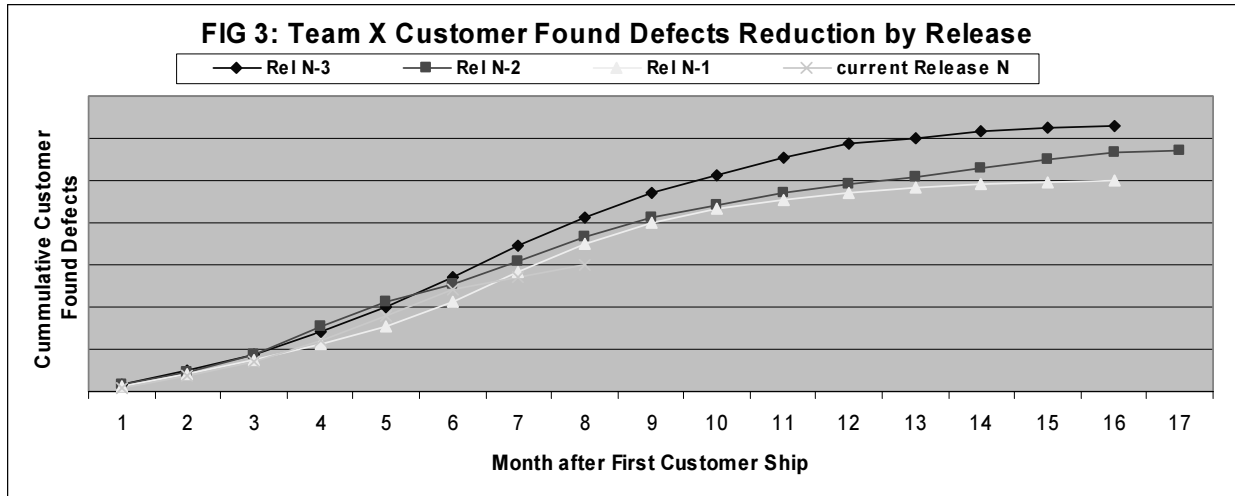


FIG 3: Team X Customer Found Defects Reduction by Release

The Success Factors

The following are the main factors that made this crash reduction effort possible:

- Leadership
- Reduction Strategy
- Involvement
- Implementing rigorous reduction actions

LEADERSHIP:

This was the most crucial factor that made the effort successful. The organization's General Manager (GM) was fully engaged in sponsoring the effort to eliminate crashes. The GM started the effort by developing and communicating the organization's vision of "Zero customer crashes" to all members of the organization. The GM identified and actively recruited the people with the right skills and enthusiasm to lead the effort. For example, a Senior Engineering Director was selected and recruited to lead the effort. The GM closely followed the progress of the crash reduction effort – giving praise and recognition to teams and individuals when significant progress was made. He raised the visibility of the effort and provided coaching and support when there was regression in performance. There were weekly discussions between the GM and his direct staff to maintain a continuous focus on the customer crash reduction effort.

REDUCTION STRATEGY:

To ensure the success of the effort, the crash reduction program was divided into two phases:

Phase 1: focused on the immediate need for reducing the current backlog of unresolved known crashes. The objectives for this phase were to bring the crash management process under control and to improve the responsiveness to crash issues.

Phase 2: focused on the longer term reduction and, if possible, the elimination of new incoming crash issues. The objectives of this phase were to identify the root causes of the crashes in order to eliminate current problems and prevent similar issues from happening in the future. An additional benefit of this effort will be as the rate of incoming crashes decreases, the team's responsiveness to crash issues and the cost of rework will also improve.

INVOLVEMENT:

Almost everyone in the organization was involved in the crash reduction effort. A taskforce was formed and named the “SEGV taskforce” (SEGV, segmentation violation, is a technical name for the most severe type of crash). This was led by a Senior Engineering Director and was chartered with the overall responsibility for the crash reduction effort. The Taskforce, in turn, worked with all the appropriate people in the Engineering group to put the crash management process under control. Engineers, Product Managers, Product Directors, Program Managers, Release Managers and Enterprise Quality Consultants were all involved.

IMPLEMENTING RIGOROUS REDUCTION ACTIONS:

For Phase 1, the focus was on setting up a project tracking and oversight infrastructure to manage crash resolutions and backlog reduction. The taskforce identified, tracked and published the information about the known crashes to all involved Engineering personnel every week. The taskforce informed the owner(s) of the code that caused the crashes and worked with them to ensure closure of the issues. Crashes that had not been resolved for more than 30 days got special attention from Senior Management.

For phase 2, the focus was on enhancing the software development lifecycle with new formal technical best practices that would reduce/eliminate new incoming crash issues. The main actions were:

- Formalize engineer testing: Each of the new/enhanced features needed to successfully complete the Unit Level Testing by the Beta test entry time. The unit test case(s) had to be documented and successfully tested. Unit Test cases and their associated test results were archived for potential integration into the regression test suite and also for later audit.

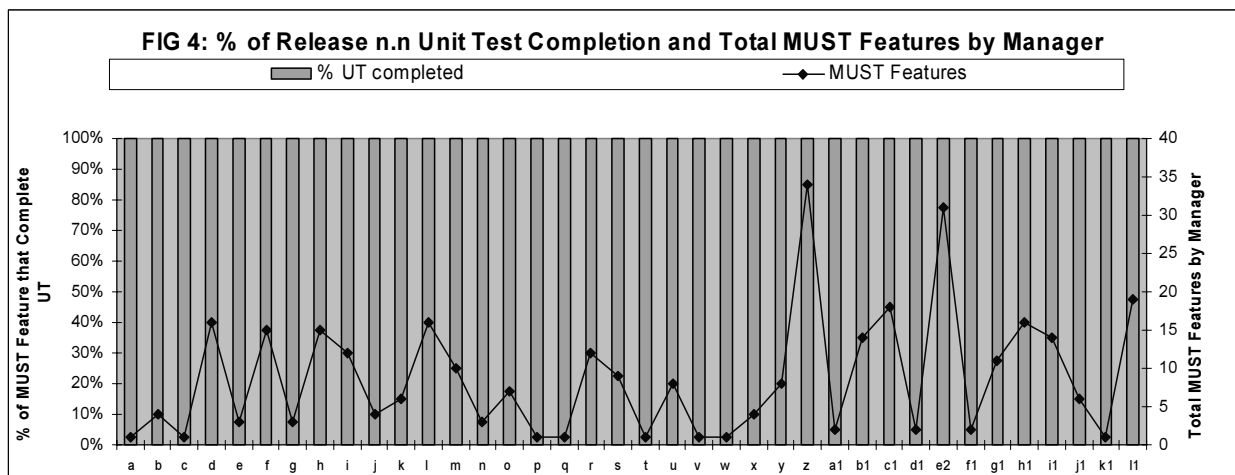


FIG 4: % of Unit Test Completion and Total MUST Features by Manager –Release nun

- Use a Static Analysis tool (QAC++ in this case) to identify potential severe issues in the code and fix them. The team worked with the QAC++ vendor Architect to identify the essential subset of the tool checking rules that identified code issues that could cause crashes and other serious code issues. This class of code issues is called severe issues. The team comprehensively implemented QAC++ in two ways. At the individual Engineer level, the tool was set-up to improve the unit code quality before it was checked-into the main code stream. At the release level, the tool was set-up to ensure compliance to the quality standards by the main code stream before it was released to customers.

- Perform Root-Cause Analysis of known crashes to identify the common problems, their common causes, how they were fixed, and what can be done to prevent these common crash issues from happening again.
- Incorporate the learning about crash prevention into future release planning so sufficient resources and time can be allocated to the necessary crash improvement activities.

The Best Practices Used

The following best practices were used by the team to eliminate potential crashes in the product:

- Identifying product crashes
- Effective use of Static Analysis tools to identify and fix potential product crashes
- Identifying the crash-prone area for resolution/reduction

IDENTIFYING PRODUCT CRASHES:

A query on the company defects database for bugs from the previous year containing “crash”, “core-dump”, and “SEGV” issues resulted in a large number of hits, of which ~20% of hits were associated with customer found problems. If we use the software industry estimate of 80 hours of effort required to find, validate, fix, and revalidate a customer found problem and about 15 hours to do same for an internally found problem, then the effort to find and fix these crash problems can account for a quite sizeable effort. This is a significant cost for the company. A similar query for just the few months before the crash reduction program started resulted in a similar proportion of hits. It appears that there wasn’t much improvement in the crash issues situation just by giving the organization more time. This was the main reason for the organization’s GM starting the whole crash reduction effort.

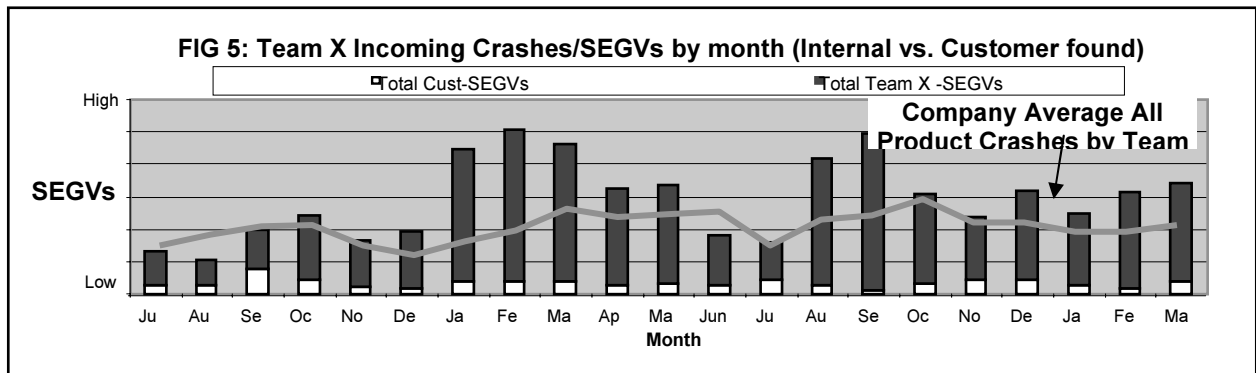


FIG 5: Team X Incoming Crashes/SEGVs by Month (Internal Versus Customer Found)

The regular (weekly) tracking of crash related metrics gave the organization the invaluable focus on resolution of the issues that eventually provided the expected improvements in crash reduction.

EFFECTIVE USE OF STATIC ANALYSIS TOOLS:

Beside product crashes, there are other serious issues that can impact customer satisfaction as well. In general, we can group these “bad” issues (crashes and other types of serious issues) together under the classification of “severe” issues. It will be fantastic if we can eliminate these severe issues from our product so we can channel the effort currently going toward fixing them into more interesting work such as new feature development. It would be even more fantastic if we can find a cheap but simple and effective way to do so. New developments in code quality analysis technology made possible the ability to find “severe” issues in the product code. Static Analysis tools are the class of application software used to analyze product code and identify potential severe issues (more than just crashes). To effectively use

static analysis tool to identify severe code issues, we will need to know what the severe problems in our products are, what the impacts of those problems on product quality are and, potentially, what we can do to eliminate them.

Severe code issues can be defined as the issues related to the abnormal termination of the code execution, the corruption of the data used, the violation of memory allocation, segment violation (SEGV), and incorrect logic implementation in the products. These issues can severely impact the correct behavior of our products and cause a negative customer experience and perception about our product quality. Typical severe issues in C++ applications and their quality impact are described in Table 1.

The current crop of leading static analysis tools on the market come with a large set of capabilities that sometimes can overwhelm even experienced software developers. It is very important in planning for the use of static analysis tools to choose a small set of tool features that are specifically aimed at the discovery of the above class of severe issues. Checking for only this subset of issues by the tool can reduce the reporting of “false positives. False positives are reports from the tool of an issue that turns out to not be an issue after all. Reducing false positives helps the tool to be better accepted by the code developers by reducing the amount of output that they need to review.

IDENTIFYING THE CRASH-PRONE AREA FOR RESOLUTION/REDUCTION:

We all want to resolve every issue causing a product crash, however, in reality, we can only resolve the issues that our resources and/or business schedule allows. We must find a way to prioritize crashes to maximize return of investment (ROI) for our crash resolution efforts. There are many ways to do this. Typical prioritization approaches that can be used to identify areas for focused improvement are: crash-prone area identification, common crash-type classification, and crash escaped analysis. Figure 6 is an example of how product crashes are tracked by product. In this example, we can see products a, b, c, and d will need more attention to resolve product crashes.

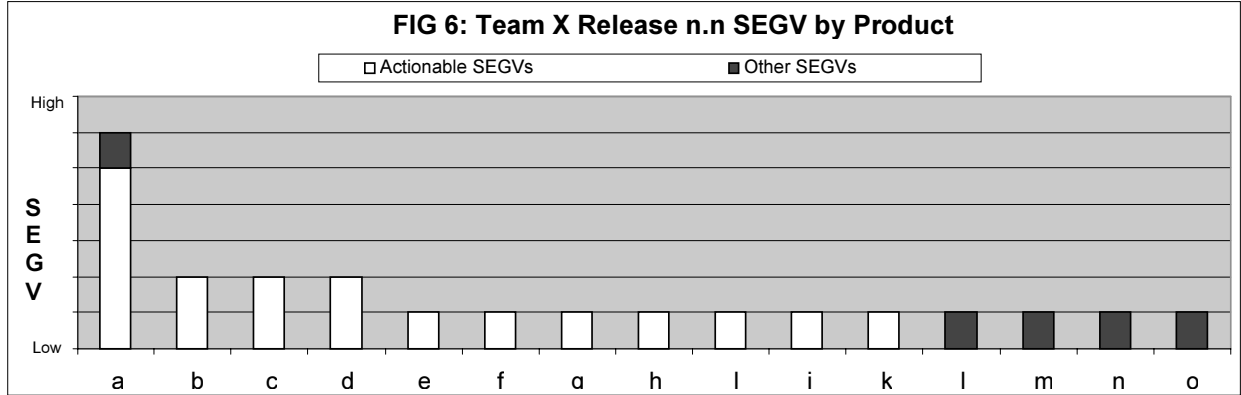


Figure 6: **Team X Release n.n SEGV by Product**

Figure 7 shows an example of crash issue classification for crash-prone product Y. In this case, the majority of the crashes fell into the error-checking category (error-checking was not built into the product code). Potential resolution for this common class of crashes is to implement the features in the product code to check for errors, such as division by zero or out-of-bound conditions, before code operations are performed.

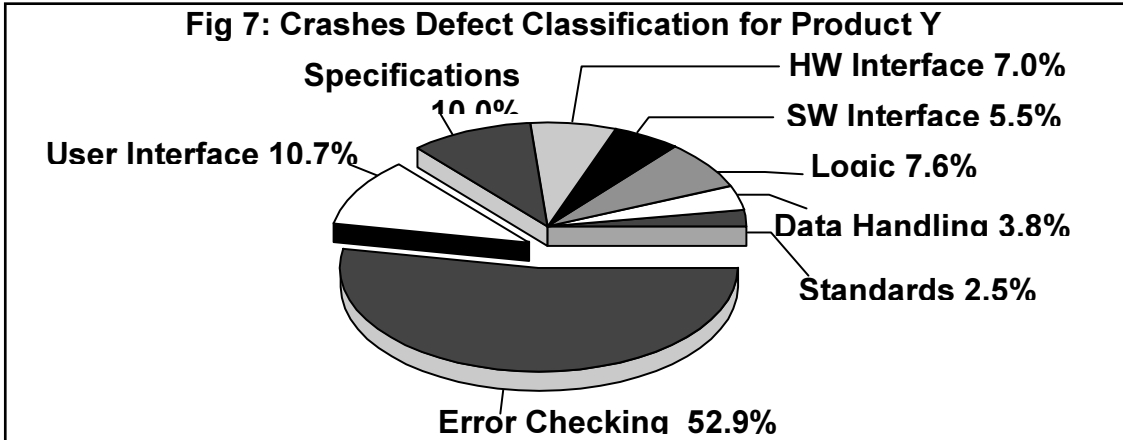


Figure 7: Crash Defect Classifications for product Y

Figures 8a through 8c are examples of crash-escaped rate analysis. Each numeric value in these charts shows the percentage of crashes found by the noted function.

In Figure 8a, a majority of crashes escaped R&D testing and were found by System testing (Operations in this case means System Testing). One potential improvement in this case is more focus on the R&D check/testing of products before turning the products over for System testing.

In Figure 8b, the case is reversed. The crashes that escaped the detection during the early phases of product development (i.e. Requirements and Architect/Design) were found by R&D. In this case, additional document and code review coverage would prevent many of the crashes that reach the R&D testing phase.

In Figure 8c, a large number of crashes escaped the internal testing effort. In this case, perhaps more flow testing or customer-like usage testing could reduce the amount of crashes that escape to customer.

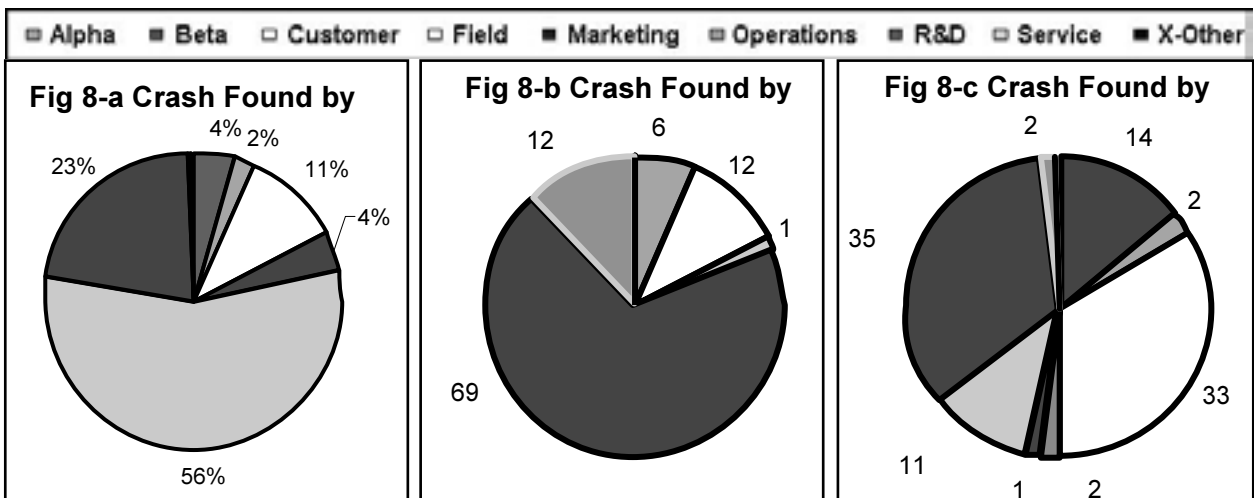


Figure 8: Examples of Crash-escaped Rate Analysis

Summary

Product crashes are annoying and can cause customer dissatisfaction and also a large amount of expensive rework. We want to eliminate and/or prevent product crashes as much as possible. This article described the process one company used to reduce the number of product crashes in an existing product line. This included: querying for product crashes among existing defect records, creating various metrics based on the identified crash reports to identify the crash-prone areas, and making the appropriate process and product improvements to prevent future crashes. Trend charts can be used to track and monitor the progress of prevention and reduction efforts.

Best practices that any team can use to ensure product crashes reduction are: leadership, participation by all members of the team, use of failure analysis and root-cause analysis to identify error areas for focused effort, and use of static analysis tools for identifying potential code issues.

TABLE 1: C++ “severe” issues

- **Abnormal Termination** (termination by C++ run-time environment)
 - Uncaught 'throw' expression in destructor -> if another exception is already being handled, and this destructor is called during stack unwinding, C++ run-time will stop - clean immediate termination
 - Throw of an exception of a type which is not listed in the exception specification -> C++ run-time will stop - clean immediate termination (for ISO C++ compliant compilers - doesn't include GCC compiler, unless -fexceptions option is used)
 - Rethrow expression is outside a catch block -> C++ run-time will stop
 - Throw in main will cause the program to terminate -> C++ run-time will stop
- **Data Corruption:**
 - Missing required return expression -> Garbage returned from function
 - The default value is different from the overridden function -> Inconsistency between the default parameter value in base and derived classes
 - Unary minus operator is being applied to an unsigned type -> The value will remain as unsigned (has the two's-complement value)
 - An implicit conversion from an array of derived to pointer to base -> if sizeof (base) < sizeof (derived) then incorrect index to the array would be used -- because of extra member variables in the derived class. If the derived class has no new additional members then it is OK (this issue can also cause SEGV)
 - Passing a class object as an ellipsis argument -> for Ellipsis arguments (i.e. third argument to printf function) the value is copied bit-wise on the stack, and no type conversions are applied (this issue can also cause SEGV)
 - Builtin object or argument is modified and accessed between sequence points -> the order of evaluation is un-specified for operands in an expression. It is not guaranteed to be left to right so the compiler can optimize the code
 - The right hand side operand of the shift operator is negative or is too large -> undefined behavior, meaning the resulting value cannot be pre-determined
 - A jump past initialization of objects -> use of un-initialized data
 - Function has a non-void return type but no return statements -> Garbage returned from function
 - Uninitialized member used as initialiser -> use of un-initialized data
- **Memory violation** (cause memory leaks):
 - Object used with inconsistent allocation methods in different translation units -> Link-time issue - inconsistent use of plain and array allocation of the same object in the project
 - There is no corresponding operator delete for this operator new -> Use of custom 'operator new' with default 'delete'
 - This object was used as pointer to non-array object earlier -> inconsistent use of plain and array allocation in the same source file
 - This object was used as pointer to array earlier -> inconsistent use of plain and array allocation in the same source file (reverse of previous case)
 - This object was used as a pointer to old C-style allocated memory -> inconsistent use of C and C++ allocation in the same source file (malloc and new) -- malloc doesn't call constructor and free doesn't call destructor unlike new and delete
- **SEGV** (segmentation violation - termination by the OS)
 - The memory referred to has been deallocated -> termination by the OS
 - Index is out of bounds -> termination by the OS
- **Wrong Logic implementation:**
 - The operand of 'sizeof' has side effects that will not be performed at run-time -> operand of sizeof is evaluated at compile-time, so any side effect will not be performed, i.e. a function call or modification to variable

- Exception is derived from an exception caught above -> this catch-clause will never be executed - dead code
- This catch-all exception handler should be the last -> the following catch-clause will never be executed - dead code.

References

Review and Inspection

- Michael Fagan - Inspections - <http://www-isd.fnal.gov/ftr/www/faganInsp.html>
- Tom Gilb - Software Inspection, Addison-Wesley Longman Publishing Co., 1993
- Karl E. Wiegers - Peer Reviews in Software: A Practical Guide, Addison-Wesley, 2002

Continuous improvement, Metrics, Root-cause-analysis:

- Watts S Humphrey:
 - Managing the software process, Addison-Wesley Longman Publishing Co, 1989
 - Introduction to the personal software process, Addison-Wesley Longman Publishing Co, 1997
- Robert R. Grady:
 - Software Metrics: Establishing a Company-Wide Program, Englewood Cliffs, NJ: Prentice Hall, 1987
 - Practical Software Metrics For Project Management And Process Improvement, Prentice Hall, 1992
 - Successful Software Process Improvement, Prentice-Hall, Inc. 1997

Learning Organization

- Peter Senge - The Fifth Discipline: the Art and Practice of the Learning Organization – Currency, 1990

Static Analysis tools:

- http://weblog.infoworld.com/article/06/01/26/73919_05FEcode_3.html (Between klockwork and coverity)
- <http://www.ep.liu.se/ea/trcis/2008/003/trcis08003.pdf> (All the tools)
- http://weblogs.java.net/blog/cos/archive/2006/10/static_analyzer.html (klockwork and coverity)
- <http://pubs.drdc.gc.ca/PDFS/unc69/p528977.pdf> (Almost all the static analyzer tools available. Has some info about Polyspace too.)