Using Defect Tracking and Analysis to Improve Software Quality

A DACS State-of-the-Art Report

Contract Number SP0700-98-D-4000
(Data & Analysis Center for Software)

Prepared for:
Air Force Research Laboratory -
Information Directorate (AFRL/IF)
525 Brooks Road
Rome, NY 13441-4505

Prepared by:
Michael Fredericks and Victor Basili
Experimental Software Engineering Group
University of Maryland
College Park, Maryland USA
fred@cs.umd.edu, basili@cs.umd.edu
and
DoD Data & Analysis Center for Software (DACS)
ITT Industries - Systems Division
Griffiss Business & Technology Park
775 Daedalian Drive
Rome, NY 13441-4909

DoD Data & Analysis Center for Software (DACS)
P.O. Box 1400
Rome, NY 13442-1400
(315) 334-4905, (315) 334-4964 - Fax
cust-laisn@dacs.dtic.mil
http://www.dacs.dtic.mil

The Data & Analysis Center for Software (DACS) is a Department of Defense (DoD) Information Analysis Center (IAC), administratively managed by the Defense Technical Information Center (DTIC) under the DoD IAC Program. The DACS is technically managed by Air Force Research Laboratory Information Directorate (AFRL/IF) Rome Research Site. ITT Industries - Systems Division manages and operates the DACS, serving as a source for current, readily available data and information concerning software engineering and software technology.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	14 November 1998	N/A

4. TITLE AND SUBTITLE
A State-of-the-Art-Report for Using Defect Tracking and Analysis to Improve Software Quality

5. FUNDING NUMBERS
SP0700-98-4000

6. AUTHORS
Michael Fredericks
Victor Basili

7. PERFORMING ORGANIZATIONS NAME(S) AND ADDRESS(ES)
Experimental Software Engineering Group
University of Maryland, College Park, Maryland and
ITT Industries, Systems Division, 775 Daedalian Drive
Rome, NY 13441-4909

8. PERFORMING ORGANIZATION REPORT NUMBER
DACS-SOAR-98-2

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Technical Information Center (DTIC)/ AI
8725 John J. Kingman Rd., STE 0944, Ft. Belvoir, VA 22060
and Air Force Research Lab/IFTD
525 Brooks Rd., Rome, NY 13440

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
N/A

11. SUPPLEMENTARY NOTES
Available from: DoD Data & Analysis Center for Software (DACS)
775 Daedalian Drive, Rome, NY 13441-4909

12a. DISTRIBUTION/ AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

12b. DISTRIBUTION CODE
UL

13. ABSTRACT *(Maximum 200 words)*

Defect tracking is a critical component to a successful software quality effort. In fact, Robert Grady of Hewlett-Packard stated in 1996 that “software defect data is [the] most important available management information source for software process improvement decisions,” and that “ignoring defect data can lead to serious consequences for an organization’s business” [Grady96]. However, classifying defects can be a difficult task. As Ostrand and Weyuker paraphrased a 1978 report by Thibodeau, defect classification schemes of that time period often had serious problems, including “ambiguous, overlapping, and incomplete categories, too many categories, and confusion of error causes, fault symptoms, and actual faults.” [OstrandWeyuker84]. Yet the classification of defects is very important, and by examining the lessons learned by other organizations, one hopes to be in better position to implement or improve one’s own defect classification and analysis efforts. To this end, this report discusses five defect categorization and analysis efforts from four different organizations. This list of organizations should be taken as a sample of the range of schemes covered in research and in industry over the previous twenty-five years. The analysis efforts at these organizations generally focus on one of three goals: finding the nature of defects, finding the location of defects, and finding when the defects are inserted, with the intent of using this information to characterize or analyze the environment or a specific development process. At the conclusion of this discussion, the results of two surveys covering the defect classification and analysis practices of approximately 30 companies are presented. These surveys give some insight into how industry is currently handling this complex issue. Finally, section III of the report presents some suggestions of how companies could begin or expand their defect classification efforts.

14. SUBJECT TERMS
Defense Analysis, Software Testing, Software Engineering, Technology Evaluation

15. NUMBER OF PAGES
27

16. PRICE CODE
N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Table of Contents

List of Figures

Using Defect Tracking and Analysis to Improve Software Quality[1]

I. Introduction

Defect tracking is a critical component to a successful software quality effort. In fact, Robert Grady of Hewlett-Packard stated in 1996 that "software defect data is [the] most important available management information source for software process improvement decisions," and that "ignoring defect data can lead to serious consequences for an organization's business" [Grady96]. However, classifying defects can be a difficult task. As Ostrand and Weyuker paraphrased a 1978 report by Thibodeau, defect classification schemes of that time period often had serious problems, including "ambiguous, overlapping, and incomplete categories, too many categories, and confusion of error causes, fault symptoms, and actual faults." [OstrandWeyuker84]. Yet the classification of defects is very important, and by examining the lessons learned by other organizations, one hopes to be in better position to implement or improve one's own defect classification and analysis efforts.

To this end, this report discusses five defect categorization and analysis efforts from four different organizations. This list of organizations should be taken as a sample of the range of schemes covered in research and in industry over the previous twenty-five years. The analysis efforts at these organizations generally focus on one of three goals: finding the nature of defects, finding the location of defects, and finding when the defects are inserted, with the intent of using this information to characterize or analyze the environment or a specific development process. At the conclusion of this discussion, the results of two surveys covering the defect classification and analysis practices of approximately 30 companies are presented. These surveys give some insight into how industry is currently handling this complex issue. Finally, section III of the report presents some suggestions of how companies could begin or expand their defect classification efforts.

The first significant work on defect classification was done in 1973 by Albert Endres of IBM in Boeblingen, Germany [Endres75]. Endres took data gathered on 432 defects discovered during the test phase of development of the DOS/VS operating system. Endres classified the defects into 6 general categories, including "machine error", "user error," and "documentation error." He also classified each defect by type. His type classification scheme was very complex, dividing each defect into groups including "dynamic behavior and communication between processes" and "Counting and Calculating" among many others. After classifying the defects, Endres set out to answer several questions about each defect, including where, when, and why the defect was inserted into the program, who inserted it, and how to determine the best procedure for detecting this type of defect. During the course of his analysis, Endres examined the types and number of defects found per module and the distribution of error types. While he acknowledged that he did not have enough data to answer most of his questions, he did manage to generate several useful conclusions, most notably that 85% of defects on this project could be repaired by making changes to only one module. This contradicted common wisdom of the time which felt that interfaces between modules would be a very large source of defects.

---

1 The writing of this report was supported by contract 1800312 between the Data and Analysis Center for Software and

The possibilities of this line of research led others to pick up where Endres left off. In 1976, the University of Maryland began a collaborative effort with the Naval Research Laboratory that used defect classification to determine whether or not the design or requirements process on a project met the goals of that process. This work continued in the early 1980's at the Software Engineering Laboratory at NASA-Goddard where defect data was used to help evaluate the impact of changes in the development process. Researchers at Sperry Univac in the early 1980's developed a fault classification scheme for the purpose of enabling statistical assessments of software. During the mid-1980's, researchers at Hewlett-Packard instituted defect classification as a portion of a large-scale company-wide measurement program, and those efforts continued through the 1990's. In the early 1990's, IBM developed two new technologies using defect data. The first, Defect Prevention, involves development teams contributing to a knowledge base containing common defects, how they can be prevented, and how to easily detect them. The second, Orthogonal Defect Classification, involves using statistical methods to predict the phase in which a defect originated, rather than relying on the subjective evaluation of an engineer. In Section II of this report, each of these efforts are discussed in greater depth.

Before continuing, we need a consistent, standard terminology to use when discussing defects. [IEEE83] describes such a terminology. A fault is injected into the system whenever a human developer makes a mistake. The mistake itself is called an error. One error can lead to multiple faults. For example, suppose a developer misunderstands a concept described in a design document, and then develops code that does not meet the design. The error is the misunderstanding, and it leads to a coding fault. Similarly, if a developer is building a system involving the use of shared resources, the developer's misunderstanding of the workings of the programming language's semaphore mechanism (the error) leads to incorrect semaphore code in several places in the program (the faults) A failure occurs whenever the behavior of a software system differs from the behavior described in the requirements for the system. The failure is the outwardly observed behavior that may indicate the presence of a fault.

Note that all faults are not necessarily caused by errors. For example, consider a careless developer who, wishing to add two variables, forgets to hit the <shift> key and accidentally types "a=b" rather than "a+b." This fault is the result of an accident, not of an underlying error. Similarly, not all faults lead to failures. If the code containing the fault is never exercised, a failure will not occur. Even if the fault-containing code is exercised, unless there is some external behavior that occurs due to the fault, there will be no failure. Finally, not all failures are caused by faults in the code. The fault could lie in the specification or design documents as well. A designer could misinterpret the requirements and inject a fault in the design. Code could be written that matches the faulty design perfectly, but a failure could still result since the system will not perform as required.

---

2 While many of the works discussed below adhered to the IEEE standard, not all did; in these cases the authors' terminology was converted to the IEEE standard. Thus, the terminology in this report may not match the terminology used in the original work. Throughout this report, the term "defect" will mean any fault, failure or error occurring in a software system.

Typically, a software organization tracks faults and failures[2] using forms such as Figure 1. The contents of the form depend on the quality goals of the organization. In the next section, we examine several current fault collection and tracking practices and present some general guidelines.

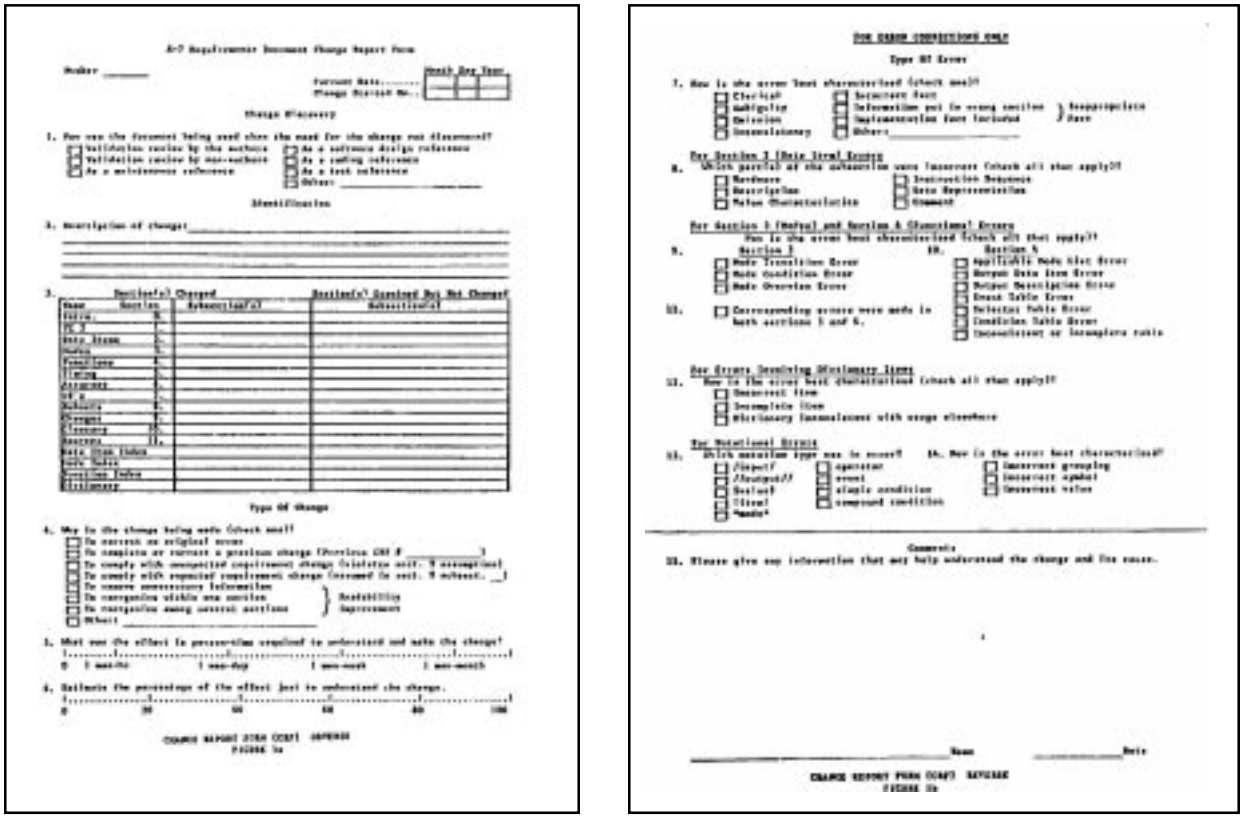


Fig 1. Sample change report from the Naval Research Laboratory [BasiliWeiss81].
This form tracked all software changes, including those due to defect correction.

II. Defect Classifications

A. University of Maryland – Naval Research Laboratory – 1976 NASA-Goddard/Software Engineering Laboratory – 1984

The goal of the researchers working at the Naval Research Laboratory (NRL) was to determine whether or not defect classification would be a successful method of evaluating a software development methodology. This work began in 1976 [Weiss78] and continued through the early 1980's [BasiliWeiss81]. The first project, [Weiss78], involved the development of a computer architecture simulator. The development team wished to determine whether the design techniques they had chosen would support three main goals of their design process. To that end they generated three questions to focus their analysis efforts:

1. Are the modularization techniques effective? The development team applied the information-hiding principle in generating their design. They broke the design system down into modules in a manner that hoped to hide design decisions that were expected to change during the lifecycle.

2. Are the detailed interface and coding specifications useful during subsequent phases of the development lifecycle? The development team spent a great deal of effort on these specifications.
3. Are the defect detection methods successful in finding defects, and more specifically can they find them early in the development lifecycle. Of specific concern to the development team on this project was an area of the product referred to as the "descriptor tables." These tables contained the state information of the architecture being simulated, so it was critical to eliminate defects in this area of the system. To achieve this goal, the development team added a rich set of error-detection routines to aid them during the testing of the system. Additionally, the development process for this project involved a code review of each module as soon as the module could be compiled.

The researchers realized that the methods of validating the hypotheses needed to be determined a priori, so as to insure that all necessary data would be collected. Once they had determined these methods (discussed below), the researchers developed specifically for this project a questionnaire called an "error report." The developers completed the two-page error reports after each defect was discovered. In order to validate the data, an error analyst would examine each error report shortly after it was filed. If the analyst had any concerns or questions about the error report, the analyst would interview the developer who filed the report. This was done as quickly as possible after the report was filed so that the incident would be as fresh in the mind of the developer as possible during an interview.

The researchers classified each defect in four ways. First, they determined the source of the misunderstanding that led to the defect. There were eight categories in this class: Requirements, Design (excluding interface), Interface, Coding specifications, Language, Careless omission, Coding standards, and Clerical. Second, each defect was classified based on the effort required to repair the defect. Easy defects required less than a few hours to repair, Medium defects required a few hours to a few days to repair, and Hard defects required more than a few days to repair. Third, each defect was classified based on how it was discovered. Finally, each defect was classified based on the number of modules analyzed in order to determine the source of the misunderstanding that led to the defect. A total of 143 defects were reported on this project.

The raw data described above allowed the researchers to answer the three questions of interest listed earlier. To determine the success of the modularization efforts, they noted that only 9 of the 143 defects (6%) resulted from interface misunderstandings. Of these 9 defects, 8 were "easy" to fix and one was "medium." Additionally, only 8 of the 143 defects required an engineer to understand multiple modules to determine the source of the misunderstanding, and only 1 defect of the 143 was created when another defect was repaired. The subjective evaluation of these results indicated that the designers successfully achieved their goal of information hiding. Additionally, this helped answer the second question about the usefulness of the detailed interface specifications. The researchers attributed some of the success in limiting interface-related defects to these specifications. The researchers also used the data on how engineers repaired defects in order to determine the usefulness of the detailed coding and interface specifications. Studying the coding specification was the second most popular technique for determining the required fix for a defect. The specifications were used in the repair of 46% of all defects. (Note multiple techniques could be used on each defect). The most popular technique was code inspection at 61%.

To answer the question regarding defect detection, the researchers noted that 43% of the defects detected during the lifecycle involved descriptor table access . The error-detection routines caught 56% of these defects. The researchers subjectively evaluated the error-detection routines as successful in catching defects. It is difficult to judge the cost-effectiveness of these routines, however the error-detection routines detected most of the defects early in the development lifecycle, and all but one of the defects caught by the error-detection routines were easy to fix.

Finally, the researchers noted an interesting correlation between the effort required to fix a defect and the time in which it was discovered. 21 of the 22 medium and hard defects were discovered during the final 10% of the development lifecycle. Nine easy defects were discovered during this time as well. While the data does not support any causality between the phase the defect is discovered and the effort required to repair it, there is a significant correlation here.

The researchers noted that most of their conclusions were subjective. They commented that it is difficult to replicate this type of work or compare work in one company or project to another because of confounding factors such as differing experiences of the development team and differing environments. Thus, statistical confirmation of hypotheses is often impossible. However, this research did show that defect classification and analysis can be a useful tool for analyzing the success of software development methodologies.

The work at the Naval Research Laboratory continued through the early 1980's. In [BasiliWeiss81], researchers proposed a new software methodology for the development of flight control software. The research focused on the requirements phase, and change data (including defect data) were used to evaluate the new requirements techniques. The researchers generated twenty-one questions of interest regarding the new techniques, several of which were defect-related. Among other things, the researchers wanted to know which sections of the requirements contained the most defects and what types of defects were contained in the document. To answer the latter question, each defect was classified as one of 8 types: Ambiguity, Omission, Inconsistency, Incorrect Fact, Clerical, Information put in wrong section, Implementation fact included, and Other. This classification scheme has proven to be useful, and in fact is still in use in current research being done at the University of Maryland [Shull98].

The researchers analyzed the data at the end of the requirements phase in order to give feedback to the developers before continuing with development. The data revealed that across the entire document, 80% of non-clerical defects were either omissions or incorrect facts. However in certain areas of the document, these percentages differed. Requirements regarding tables were important in this project for similar reasons as the previous project at the NRL, so the researchers separated table-related defects from the remainder of the defects and noted that while 49% of the defects in the document as a whole were classified as incorrect facts, only 29% of table-related defects fell into this category; meanwhile inconsistencies, which accounted for 13% of defects in the entire document, accounted for 21% of table-related defects. Taken out of context, these results mean little, however they were useful to the team at the time for evaluating the new requirements methodology.

Collaborative efforts between the University of Maryland, NASA-Goddard, and CSC, as the Software Engineering Laboratory (SEL), began in the mid 1970's [BasiliZelkowitz78]. One of the original focuses of the SEL was to collect as much data as possible to help analyze software development by discovering which parameters of development could be isolated and attempt to evaluate which of these parameters could be altered to improve the development techniques. From the earliest days of the SEL, defect data has been collected, and this effort continues to the present day.

During work at the SEL, researchers developed a goal-oriented methodology for data collection called the “Goal-Question-Metric” methodology (GQM) [BasiliWeiss84]. The first step of a data collection effort using GQM involves setting the goals of the effort. These goals enable the researchers to generate questions of interest that would address the goals and then determine the specific metrics that would answer the questions of interest. The result is a focused effort where excess data does not get collected, conserving valuable resources. Likewise, specifying the metrics in advance allows the research team to ensure that all necessary data is collected. The GQM methodology has been used continuously since its development, and it forms the basis for the first steps of the suggested process for implementing or improving defect analysis efforts presented in section III. The GQM methodology is discussed in much greater detail in section III.B.

The primary goal of the measurement program at the software engineering laboratory (SEL) at NASA-Goddard is to understand process, cost and reliability for the purpose of understanding the impact that changes will make to the process. When testing a new process at the SEL, a prediction is made in advance of how the project using the new process will differ from the SEL baseline. The goal of the early defect classification efforts at the SEL was to generate a baseline defect profile for projects in the lab. When a project uses a new process, the defect profile of that project can be compared to the baseline, and these measurements will help members of the SEL determine whether or not the process change was effective.

By 1984, The SEL had already developed a classification scheme used by projects throughout the lab [BasiliPerricone84]. The engineer identifying the error was responsible for classifying it in one of Basili and Perricone’s categories:

- Requirements incorrect or misinterpreted
- Functional specification incorrect or misinterpreted
- A Design error which spans several modules
- A Design error or an implementation error in a single module
- Misunderstanding of external environment
- Error in the use of programming language or compiler
- Clerical Error
- Error due to previous miscorrection of an error

In addition to the classification of the error and information describing the error and how it was found, the engineers tracked the effort needed to correct each error.

In the early 1980s, researchers at the SEL wanted to analyze errors within the particular arena of code reuse. They studied a specific project containing approximately 90 KLOC of software used in satellite planning studies. Since the SEL already had error data collection in place, it was possible to compare the results from this project with other projects in the lab to see how much this project differed from the norm. In general, the presence of a company or organization-wide program such as is present in the SEL or Hewlett-Packard is advantageous for this very reason. Additionally, in the SEL, project teams are constantly trying new processes. Before trying a process, the researchers make predictions of how the project should improve from the baseline. These hypotheses can be validated or rejected based on actual performance of the process. Building a baseline is key for a mature development environment because each environment is unique, making it unwise to use data gathered by other organizations.

The researchers in the SEL also place a high priority on validating the error data that they collect. In all projects in the SEL, the data collection efforts involve a two-step validation process. First, each supervisor is expected to sign off on each change report form after the engineer completes the change. The supervisor reads over the form and checks the data reported before signing off. Second, at the end of the project cycle, the researchers examine all the change report forms to see if they can detect any contradictions or inconsistencies. In this project, the only statistically interesting result from this second review was that errors due to miscorrection of an error were under reported, most likely due to the engineers' not being aware that other engineers had previously tried to correct the error they had just repaired.

Since the analysis of this project focused on code reuse, the researchers divided the errors found into two additional categories based on the module(s) in which the error was found. An error could be found in a "new" module, involving code generated specifically for this project, or a "modified" module, involving reused code.

When the researchers compared the types of errors found in modified modules to types of errors found in new modules, they discovered that the only significant difference is that modified modules tended to have a higher incidence of errors involving incorrect or misinterpreted functional specifications. The researchers concluded that this result was probably due to the fact that while the reused modules were taken from a different system that used similar algorithms, the functional specification was not defined clearly enough for use in the current project.

The researchers also compared the effort required to fix errors found in new modules to the effort required to fix errors found in modified modules. Fifty-four percent of the errors found in modified modules required one day or longer to fix, while only 36% of the errors found in new modules required one day or longer to fix. Additionally, a higher error density was noted in modified modules. These conclusions allowed future development teams in the SEL to weigh the higher error repair costs for modified modules against the savings in development time to determine the level of reuse that the project wished to undertake.

Lastly, the researchers noticed that the error distribution by type for this project was very different from the baseline. Twenty-two and a half percent of errors in this project were "design of a single component" errors, where the baseline indicates that typically 72% of errors are of this type. Additionally, this project showed an abnormally high percentage of errors due to misinterpreted or incorrect functional specification. Forty-four percent of the errors in this project were due to this, compared to only 3% on the typical project. This example shows how even large deviations from the baseline do not necessarily signal problems in the process, and that data from a project cannot be analyzed in a vacuum. It turns out that this type of system had been implemented previously, and the developers came to the project with a very good understanding of the design, so the ratio of design to functional specification errors was different.

B. Sperry Univac – "Attribute Categorization Scheme" – 1984

In 1984, researchers Thomas Ostrand of Sperry Univac and Elaine Weyuker of New York University noted the difficulty of conducting statistical assessments of software development processes and validation efforts due to a lack of usable data [OstrandWeyuker84]. To address this situation, the pair

analyzed software errors from a project at Sperry Univac, and used this error data to generate a fault classification scheme.

To get the error data needed to generate their scheme, the researchers asked software engineers to fill out a change report for every change made to the software being studied. The two-page form asked the engineer several objective questions including: the reason the change was requested, how and when the problem was detected and isolated, whether or not the engineer had originally written the code being repaired, how much time was spent isolating and correcting the problem, and how many modules were changed. Also asked in the form were two subjective questions: when the problem was created, and why the engineer believed the problem occurred. These change forms were then analyzed and a list of attributes generated which could be used to classify future defects.

The researchers named their method an "attribute categorization scheme" because of similarities they noted between the values that were assigned to defects and the attributes of a relation in a relational database system. Each defect is classified in four distinct areas: "Major Category," "Type," "Presence," and "Use." There are a small number of possible values for the classification in each area.

- Major Category. Used as the primary classification for the defects. Possible values include "Decision," where the fault occurs in code involving branching, "System," where the fault occurs not in the product itself but in the operating system, hardware, or other portion of the environment, and "Data Handling," where the fault lies in code that initializes or modifies data. Other values: "Data Definition," "Decision and Processing," "Not an Error," (when no change is required) and "Documentation."
- Type. Only relevant if the Major Category indicates that the fault is somehow related to data or a decision. If the fault involves data, the type attribute describes the data involved. Possible values include "Address," where the fault involved a memory access issue such as a bad array index or a pointer error, "Control," where the data involved is used for flow control, and "Data," where information was processed, read, or written incorrectly. If the fault involves a decision, there are two possible types: "Loop" and "Branch," indicating what type of decision was involved.
- Presence. A fault can either be "Omitted," where a lack of code was the cause of the fault, "Superfluous," where extraneous code included in the product led to the fault, or "Incorrect," where the code that was in the product needed to be altered in order to repair the fault.
- Use. Relevant only if the fault involves data handling. Possible values are "Initialize," "Set," and "Update," depending on the dependency between the data being handled and the value to which it is assigned.

The researchers acknowledge that since a relatively small number of errors were used to generate the scheme, the possible values for each area do not necessarily include all possible values for all possible faults. However, as new fault types are discovered, minor changes should be all that is needed for the scheme to be able handle the new faults. The classification scheme is oriented towards code faults, since the majority of the errors they tracked in order to generate the scheme resulted in code faults. The goal of the scheme is to answer the questions "What is the fault?" and "How did the fault occur?"

After developing the scheme, the researchers classified faults that were reported for this product starting shortly after unit testing had begun. When an engineer discovered some problem with the product, the engineer filed a problem report. The problem report included a description of the symptoms of the problem, a description of the problem in the code, and the modifications done to the code in order to fix the problem. The researchers took the problem reports and classified the defects themselves. They then looked at the data to see what they could learn about the product and the processes in their environment.

The researchers used the data to make many interesting observations. For example, the most common type of defect was "data definition" (32%), while "documentation" defects (1%) were relatively rare. The majority of "data handling" faults were discovered in unit test, whereas over 80% of "data definition" faults were discovered in functional testing. Fifty-four percent of faults were faults of omission. Fifty-four percent of faults found in unit test were isolated in one hour or less, but that figure jumped to 86% of faults isolated in one hour or less when they were found during functional test.

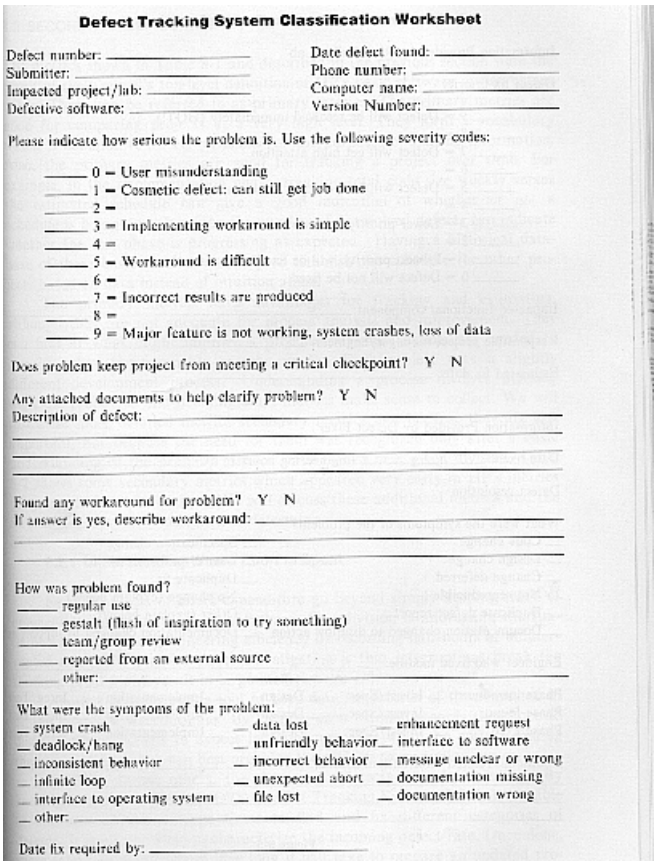
By themselves, these data are not of much interest to other organizations. The data are likely environment-specific, and quite possibly even product-specific within their environment, although Ostrand and Weyuker did find that their data corresponded roughly with some other empirical studies of the day. The specific results are not what is important. The significant conclusion is that this (or a similar) type of fault classification scheme allows an organization to develop a baseline from which one could objectively determine the effects of a process change by comparing the results of a product developed with the new process to the baseline. Additionally, the data could guide the organization by identifying problem areas where processes are in need of improvement.

In summary, the researchers at Sperry Univac wanted to develop a fault classification scheme that would allow them to perform meaningful analyses of a product or a development process. While this scheme was developed in a narrowly-focused environment, the researchers succeeded in demonstrating the feasibility of using fault classification for these purposes. That is, the specific results from their environment are probably not directly transferable to other environments, but other organizations could use a similar method to judge the effectiveness of process improvements in their environment.

C. Hewlett-Packard – "Company-Wide Software Metrics" – 1987

In 1986, Robert Grady and Deborah Caswell of Hewlett-Packard released a book which discussed efforts at Hewlett-Packard to develop a company-wide software metrics program [GradyCaswell87]. By metrics, Grady means any direct measurement made of the software, or any value that can be calculated from direct measurements (defects per KLOC, for example). While HP's efforts are not limited to defect classification, defect classification does play a significant role in their overall program. Throughout Grady's work, the term defect is used exclusively, rather than fault or error. It can be inferred that at Hewlett-Packard, "defect" is a term used to capture faults, failures, and errors.

The engineers at Hewlett-Packard file a defect classification form in the defect tracking system for each defect discovered. The form, shown in Figure 2, asks for information on the severity of the defect, the method of discovering the defect, and the symptoms of the defect. A second form, not shown, is filed in the defect tracking system to track the resolution of the defect. This form includes information on the priority of the defect (i.e. how soon it needs to be fixed), how the defect was fixed, and who fixed it. The form also asks the repairing engineer three more questions: when the defect was fixed, when the defect was found, and when the defect was introduced. While the first two of these questions are objective, the latter is subjective.



Defect Tracking System Classification Worksheet

Defect number: _____ Date defect found: _____
Submitter: _____ Phone number: _____
Impacted project/lab: _____ Computer name: _____
Defective software: _____ Version Number: _____

Please indicate how serious the problem is. Use the following severity codes:

_____ 0 = User misunderstanding
_____ 1 = Cosmetic defect: can still get job done
_____ 2 =
_____ 3 = Implementing workaround is simple
_____ 4 =
_____ 5 = Workaround is difficult
_____ 6 =
_____ 7 = Incorrect results are produced
_____ 8 =
_____ 9 = Major feature is not working, system crashes, loss of data

Does problem keep project from meeting a critical checkpoint? Y N

Any attached documents to help clarify problem? Y N
Description of defect: _____

Found any workaround for problem? Y N
If answer is yes, describe workaround: _____

How was problem found?
_____ regular use
_____ gestalt (flash of inspiration to try something)
_____ team/group review
_____ reported from an external source
_____ other: _____

What were the symptoms of the problem:

__ system crash	__ data lost	__ enhancement request
__ deadlock/hang	__ unfriendly behavior	__ interface to software
__ inconsistent behavior	__ incorrect behavior	__ message unclear or wrong
__ infinite loop	__ unexpected abort	__ documentation missing
__ interface to operating system	__ file lost	__ documentation wrong

__ other: _____

Date fix required by: _____

Fig 2. Hewlett-Packard Defect Classification Worksheet [Grady87]

Before starting to track any software metric, including data about defects, Grady emphasizes that the company or project team involved must have a clearly defined goal. Why the team wants to track defect data directly affects the specific data that are tracked. With the goal in mind, the team or company can determine the exact data to be collected. For example, in order to achieve the goal of determining the cause of defects in order to improve their development processes, HP engineers felt that they needed to determine when a defect was found and when it was fixed as stated above. Similarly, if the goal were to determine the effects of reuse on defect repair efforts (as will be discussed later in this report), effort required to fix each defect would also need to be measured. An additional benefit of setting a well-defined goal is that the focused effort can keep costs down as engineers spend no time gathering needless information.

As previously stated, the goal of the defect tracking portion of the Hewlett-Packard metrics program is to determine the cause of defects in order to improve development processes. By 1992, the defect tracking had evolved, and in [Grady92] the classification scheme shown in Figure 3 was in use.

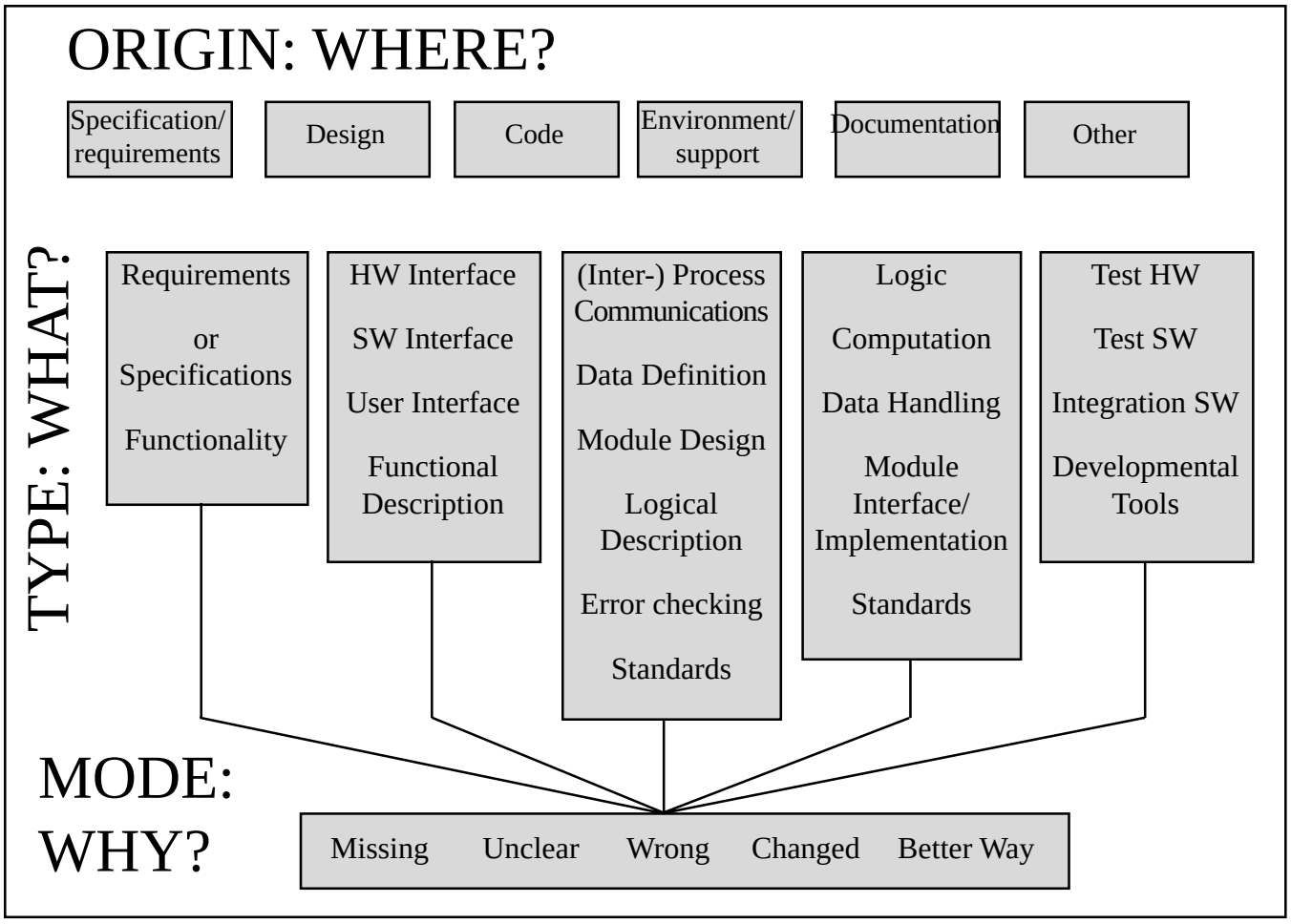


Fig 3. Hewlett-Packard Defect Categorization Scheme [Pfleeger98]

As seen in the diagram, Hewlett-Packard categorizes defects first based on the phase in which the defect was introduced. Depending on the phase, there are different types of defect, as shown in the middle layer of the diagram. All defects regardless of origin are further classified based on the mode of the defect. For instance, a code defect where error checking had been omitted would be classified under "Missing." A Design defect where the Functional Description was incorrect would be classified under "Wrong."

To achieve their aforementioned goals, the development teams examine the types of defects that occur most frequently, and the number of defects which occur in each module. The latter measure helps managers identify modules which may require extra testing or even a major rewrite. Additionally, the teams examine the phases in which defects are introduced. The data are plotted on graphs. If the only data available are the data from a single project, the development teams may simply look at the graphs to try to identify trends. If data from several projects in the environment are available, then the data from the current project can be compared to the baseline and a more formal statistical analysis can be run.

Another way to use these data for process improvement is to identify common causes of errors. One group of developers at Hewlett-Packard felt that they did not understand how defects influenced their process [GradyCaswell87]. Their measurements indicated that some projects in the group had pre-release defect densities as low as 0.4 defects per thousand lines of non-commented source code, while

others had densities as high as 6 defects per thousand lines of non-commented source. The group wanted to try to predict defect densities better and to look for ways to improve its processes to reduce the defect densities. The group began classifying defects in a similar scheme to the one showed above, but tailored slightly to fit their focus, which was developing compilers. What the group discovered was that most of the defects were injected during the detailed design phase. Not just that, but over half of the total reported defects were injected during redesigns. The group noted that the formal reviews in place for designs were not being used following redesigns. Consequently, redesign reviews were added to try to catch these redesign defects before they became faulty code.

D. IBM – "Defect Prevention" – 1990

The goal of the Defect Prevention efforts at IBM, as developed by R.G. Mays *et al*, is just what one would expect from the name. The program focuses on eliminating the introduction of defects [Mays90]. To do this, faults are analyzed in order to understand them better, which the goal of not allowing similar faults to enter the product in the future.

The cornerstone of the process is causal analysis. Rather than rely on a single developer's opinion of the cause of the fault, each developer is expected to track objective data about each fault. The fault data are entered into a database, and at the end of each phase, a meeting is conducted to analyze the faults collectively to try to determine their causes. The "causal analysis meeting" is a two hour meeting occurring at the end of every development phase. Management is not invited to these meetings in the hope that developers will be inclined to speak more freely about root causes. The development team as a whole discusses all of the faults tracked in the database. Causal analysis involves abstracting the fault to its associated error to get a better understanding of the fault. When the errors have been abstracted, they are classified into one of four categories.

- Oversight. The developer did not completely consider some detail of the problem. For example, a case might have been missed or a boundary condition not handled properly.
- Education. The developer did not understand the process or a portion of the assigned task, presumably because of a lack of training in that area.
- Communications failure. Information was not received, or information was miscommunicated and thus not understood properly.
- Transcription error. A typographical error or other mistake was made by the developer. The developer fully understood the process to be applied, but made a mistake.

The researchers feel that, in order to improve processes, more information is needed than what can be provided by simply looking at statistics on defect or error classifications for a project. So while the error classification scheme is valuable, it is just a step in the analysis process. The participants in the causal analysis meetings discuss the errors among themselves and try to determine the cause of the error and when it was introduced, with a focus on how to prevent similar errors from occurring again. Part of the process involves searching for similar errors that may still exist in the artifact but have not yet been detected. The goal of this last effort is what the researchers have named "defect extinction" – removing all existing instances of the error and adjusting the development process so that no more instances of this error will ever occur.

The last thirty minutes of the causal analysis meeting are spent examining the error information previously discussed. The purpose of this phase of the meeting is to try to identify trends in the errors that have been introduced into the product and examine the development process of the previous phase.

Portions of the development phase that had a positive impact are noted so that suggestions can be passed to other teams. Negative portions of the phase are identified, along with possible suggestions for improvements. At the end of the meeting, all of the suggestions from this last phase are logged in the action database, along with the causes for all of the errors discovered during the phase.

The next step in the defect prevention effort involves the "action team." The size of the action team is a function of the size of the software development staff; typically there is one member on the action team for every 10-20 developers on staff. The members of the action team are full-time members of the development staff who devote part of their time to being on the action team. The action team holds regular meetings to evaluate all of the suggestions that come from the causal analysis meetings. The action team decides which suggestions to implement, how to implement them, and who will handle implementing them. Some suggestions involve developing or enhancing a tool. Others involve changing a process or getting training for personnel. The action team publishes results from all of their actions in a publicly-accessible repository, in addition to highlighting key results in the organizational newsletter.

Feedback is a very important part of the defect prevention process. First, developers need to know that management takes their comments seriously and that they can influence changes in the development environment. Second, the developers need feedback to keep them up to date on current process changes and on what errors other developers discover, so that they do not make the same mistakes themselves. To serve this end, at the start of each development phase, the technical lead holds a "stage kickoff meeting" for the team. At this meeting, the lead discusses the process for the stage so that each developer knows how to perform the tasks required in the phase. The inputs to and outputs from the stage are also discussed, as is the "common error list." This list is a compilation of errors which occur frequently during this stage of development. Finally the scheduling details for the phase are laid out.

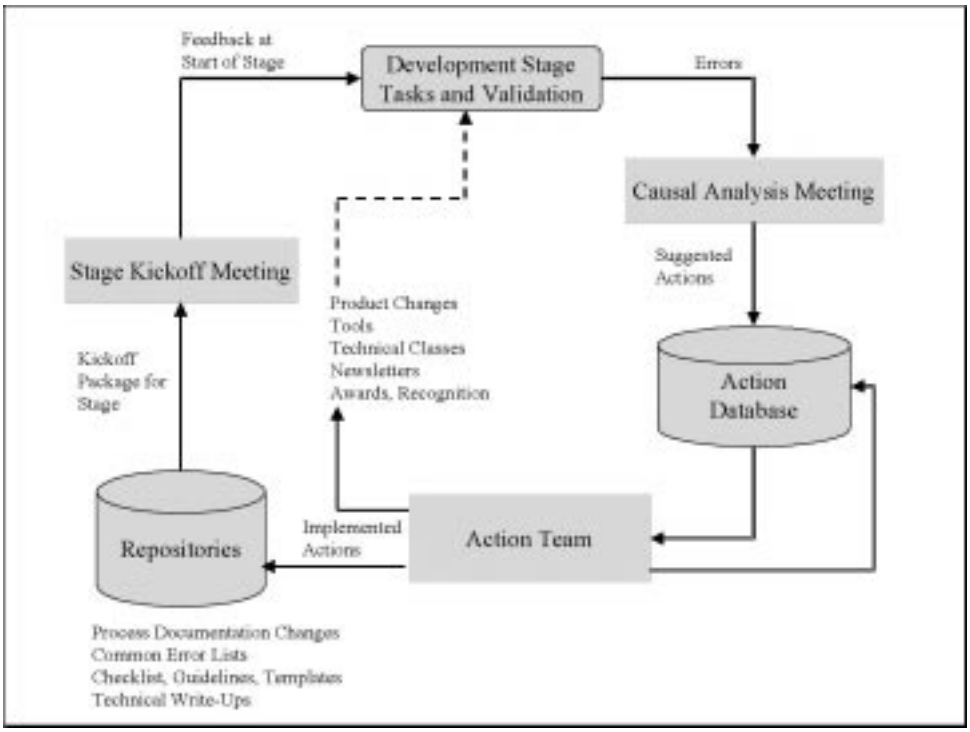


Fig 4. A flowchart of the defect prevention process [Mays90]

The researchers report promising results for defect prevention. They analyzed the faults per thousand lines of new and changed source code over the course of eight releases of the same product. The defect prevention process was introduced during the seventh release. The data reported in the seventh and eighth releases showed a reduction of faults per KLOC of approximately 50% when compared to the first six releases. The researchers estimate the costs of the defect prevention effort, including meetings attended and implementation of action items, at between 0.4% and 0.5% of total development resources.

While the above discussion focuses on defects in software artifacts, other efforts could benefit from defect prevention as well. For example, problem reports from customers could be fed into a database and the test team could analyze these reports to find flaws in the test processes and eliminate them, although an error-classification scheme tailored for these types of errors would be necessary.

E. IBM – "Orthogonal Defect Classification" – 1992

A second scheme developed by IBM, called Orthogonal Defect Classification, was developed with the goal of classifying defects so as to be able to give useful feedback to developers and managers on the progress of the current project [Chillarege92]. The researchers also wanted to develop a classification that was simple and relatively free from human error. IBM was looking for a way to predict software quality from the defect data without relying on existing statistical models. They wanted to obtain cause-effect data, where they could try to find the root cause of a defect, without relying on the qualitative opinions of the developers or testers on the team as previous classification schemes had done.

IBM chose to tackle this problem by tracking two additional attributes of a defect: the defect type and the defect trigger. The defect type attribute attempts to categorize a defect by the semantics of the repair needed to remove the defect from the system. If the fix to a defect involves changing the initialization of a variable, or a change to just a few lines of code, it is an *assignment* defect. If the fix is related to interactions between modules or device drivers, it is an *interface* fix. If the developer notices that an entire algorithm needs to be changed because it is incorrect or inefficient, the defect is classified as an *algorithm* defect. For each of these types, the developer also indicates whether the defect is one where information in the artifact is missing, or information is incorrect. All in all, there are eight possible classifications that span all the possible defect types. Eight may sound like a small number, but one of IBM's goals was to keep the number down. With fewer choices for any defect comes a greater likelihood that the developer will be able to choose accurately among the types. Additionally, the defect classes need to be distinct, or orthogonal, so that developers will not find themselves in the situation of trying to choose between types, possibly misclassifying a defect. Finally, the defect types must span all possible defects, so the developer will not be faced with a defect that does not fit easily into one of the categories. IBM originally ran a pilot study using five classes, after which the categories were increased to the current eight.

While engineers simply track the repair via the defect type attribute, the goal is to use this data to determine where the fault was injected into the system. Prior to this research at IBM, there were schemes in existence that asked the developer for an opinion on this very question, and asked the developer to classify defects based on when the defects were injected. See section C above on Hewlett-Packard for an example. The downside of those methods is that they require developers to conjecture about what occurred at some point in the past, rather than answering a question which has a definitive, easily measurable answer. The researchers found that each defect type tends to be injected in relatively

few areas. For example, assignment defects tend to be injected in the coding phase, where algorithm defects tend to be injected in the low-level design phase.

At the end of each development phase, IBM charts the number of defects of each type found in a given phase. The change in the distribution of types from phase to phase gives the development team feedback on the development process up to this point. For example, algorithm defects tend to be injected in low-level design. If the phases of development were design, code, unit test, integration test, system test, for example, one would expect a higher number of algorithm defects found in the code and unit test phase, and a steadily decreasing number over the integration and system test phase as the defects are found and fixed. If the curve is not as expected, then perhaps the development effort is not logically ready for the current physical phase.

Using the defect type classification involves making logical inferences such as the one above to try to analyze the current development process. Then data are gathered from several projects for calibrating the technique and tailoring it to a specific environment. The resulting baseline would allow for a numerical or statistical analysis of the defect type distribution, so the managers can do more than simply look for trends.

One example of using baseline data to build a model of an environment is what IBM calls “Process Inferencing.” The researchers noticed that since each defect type tends to be injected at a specific phase, the following phase should show a peak in the discovery of that type of defect. Similarly, valleys should precede and follow the peak of the discovery of the defect. A deviation from this expected trend probably indicates some deficiency in the process. In an effort to quantify this deficiency, IBM analyzed data from projects and generated a binary tree for each defect type. The tree allows the development manager to infer information about the development processes up to the current phase, hence the term “Process Inferencing.”

Each level in the “Process Inferencing” tree corresponds to a development phase. For example, the root of the tree may be “High Level Design Inspection.” The second level of the tree would then be “Functional Verification Test,” because this is the next development phase. The next level of the tree would correspond to the next phase, and so on, with the bottom level of the tree being the phase after which the development manager would like to make inferences about the development process up to that point. It is conceivable then that there would be many different models for an environment, each with different bottom levels, so that the manager may obtain feedback at multiple phases. Each branch of the tree corresponds to a number of defects of the given defect type found in that phase, either “high” or “low.” An example of this tree in [Chillarege92] for the function defect type is shown in Figure 5.

Each leaf node in the binary tree corresponds to some inference about the development process for the current project. For example, if a high number of function defects were discovered in the High Level Design Inspection, a low number of function defects were discovered in the Function Verification Test (FVT) and a high number of defects found at the system test phase, it means that the design is still exposed and that the Function Verification Test team did not do a very good job, leaving the System Test team to discover the function defects.

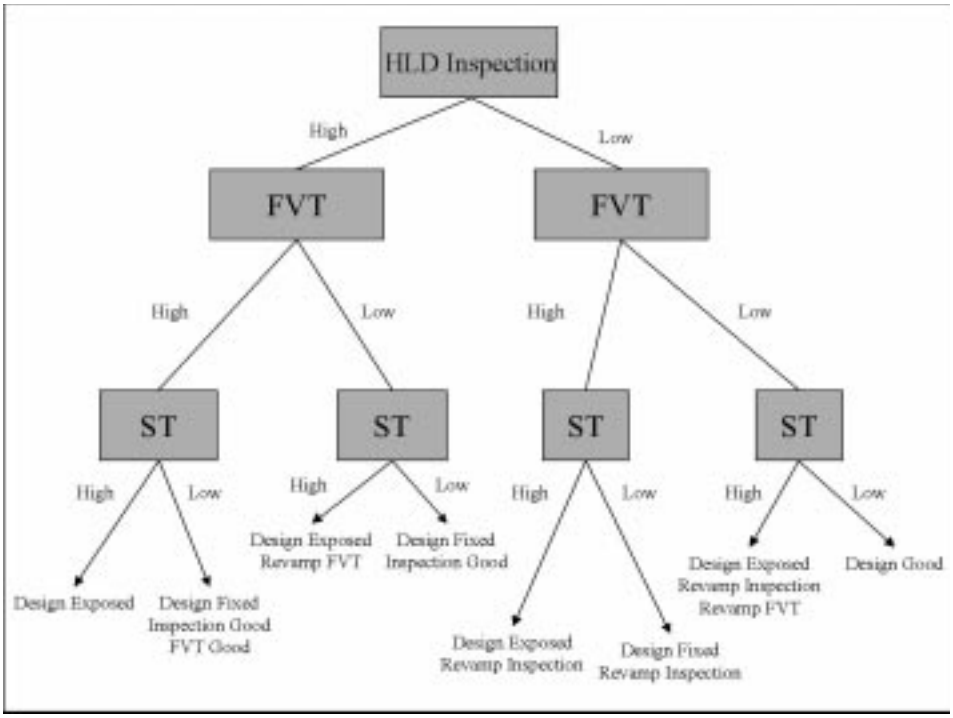


Fig 5. Process Inferencing tree for Function Defects [Chillarege92]

All of the above information can be gathered by tracking a single attribute, the defect type. The other attribute that IBM tracks is the "defect trigger," the condition that leads to the failure which exposes the defect. Again, there are a small number of triggers which are orthogonal and span the set of possible triggers. For example, a failure may be triggered by a boundary condition, a timing issue, or an exception handler. While the defect type gives feedback on the development process, the defect trigger correspondingly gives feedback on the testing and verification processes. If the defect trigger distribution after product release differs greatly from the defect trigger distribution during the system test phase, perhaps the test process is not focusing on the correct types of tests. An example given in the paper involves main storage corruption defects in a database system. The test group believed that timing issues were the most common cause of main storage corruption, and thus focused their testing efforts on timing. When the product was released, only 12.4% of failures were triggered by timing issues. The largest number of failures were triggered by boundary conditions. The test team used this information to refocus their efforts.

Defect triggers also apply to other artifact inspections, not just finished code. The researchers claims that certain defects tend to be found by inspectors with certain backgrounds. For example any inspector in a code inspection should be able to find defects triggered by lack of design conformance. Only a very experienced inspector would be expected to find defects triggered by rare situations, and inspectors very familiar with the development history of the current product would be expected to find defects related to backwards compatibility issues. After building a chart listing the possible defect triggers and the profiles of the inspectors expected to find defects with those triggers (shown in Figure 6), the development manager can use this chart to assign inspectors to the inspection task, minimizing the resources needed to cover all bases.

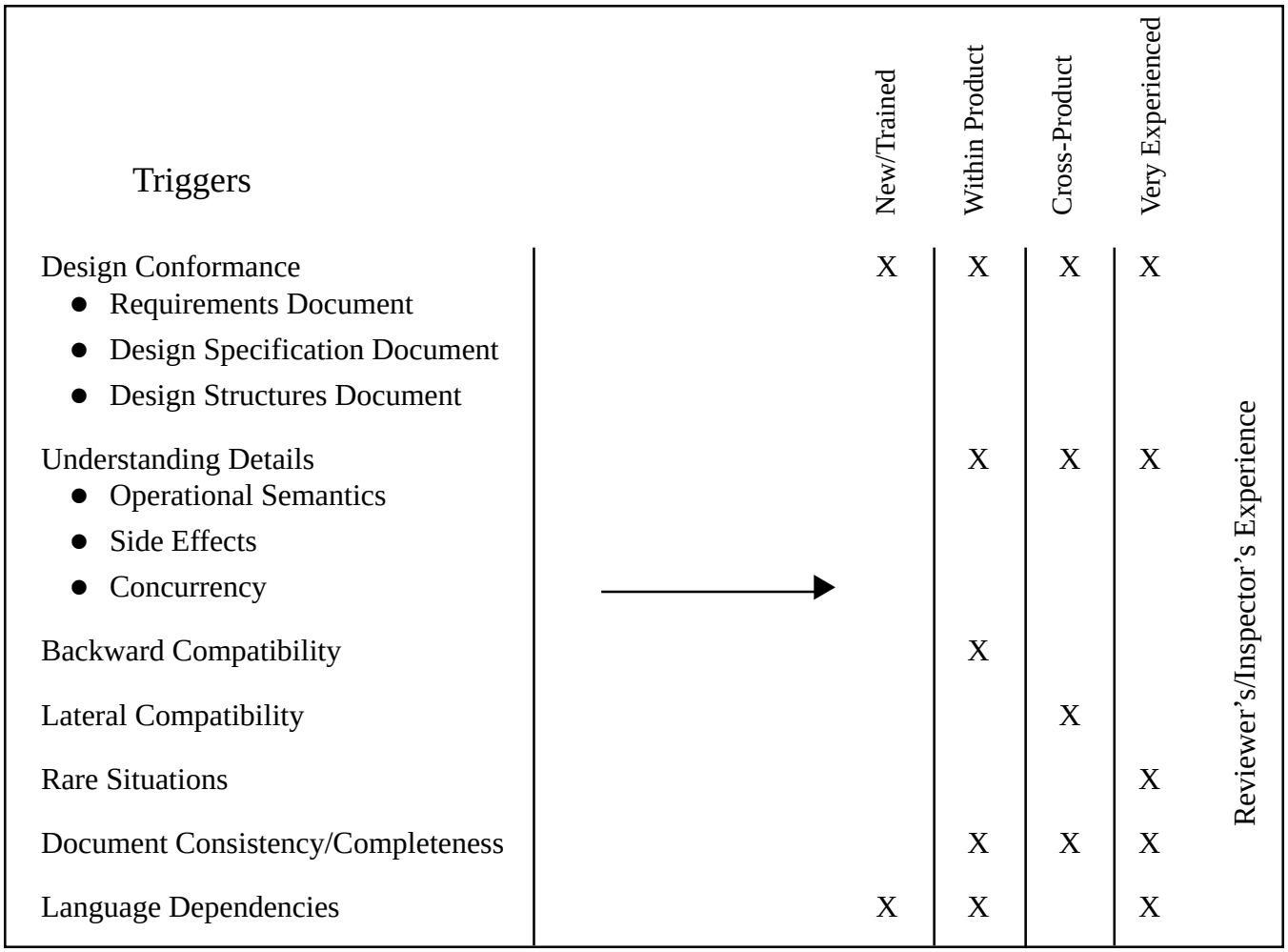


Fig 6. Chart matching abilities to defect triggers found in inspections [Chillarege92]

Additionally, for documents that are inspected multiple times, one can examine the trigger distribution of defects reported after each review. Just as the defect type distribution is expected to change from phase to phase, the defect trigger distribution should change from review to review. Once a baseline of expected trigger distribution trends is built, deviations from this expected trend can be measured, and feedback can be provided to the team working on the inspection process. For example, if defects with a certain trigger are not being found, perhaps the skill set of the inspection team needs to be re-examined, based on the chart described above.

In conclusion, IBM's Orthogonal Defect Classification scheme attempts to use easily obtainable data to make inferences about the development process. Rather than rely on qualitative input from the development team, the team members are expected to answer two relatively simple questions about each defect: the defect type and the defect trigger. Since there are relatively few possible answers to the questions asked and the answers are orthogonal and thus do not overlap, the effort to answer the questions should be minimal. Even without a baseline, a project manager can use logic to evaluate the distributions of defect type and defect trigger from phase to phase. To make deeper observations, data from several projects must be gathered to generate an organization-specific baseline to which future projects can be compared.

F. Two Empirical Surveys – 1995, 1998

In order to try to better characterize the defect classification efforts in industry today, researchers at the University of Maryland conducted two small-scale surveys on the topic. The first survey was conducted by Jyrki Kontio in 1995, where respondents were asked what defect data their organization tracked (faults, failures, errors, etc.), which types of defects they tracked, who in the organization uses the defect data, how the organization uses the data and how the data are analyzed. The questions were all multiple choice. Twenty-one companies responded to this survey.

The second survey, from the Summer of 1998, asked similar questions. However, no choices were given to the respondents. Instead, the respondents were asked to write a short paragraph describing their efforts. This free-response format elicited responses that gave more qualitative information about the processes. Questions specifically asked included: which problems the organizations are addressing with their defect tracking and analysis efforts, what types of analyses they perform, and how widely in the organization the results are shared, in addition to what data the organizations track. Nine companies responded to this survey.

The small sample sizes of the two surveys preclude statistical significance and gross generalization. However, their findings provide some useful insight into current practices. In the 1995 survey, almost 75% of the companies surveyed used defect data for process improvement, while nearly all respondents used the data to assess project status. By contrast, the 1998 survey showed minimal use of data for anything other than problem tracking or crude identification of “problem” areas of the current project which were turning up a proportionally greater number of defects than other areas. Which picture of defect data usage is accurate? The participants in the 1995 survey were also participants in the 1995 International Workshop on Software Engineering Data, which focused on using defect and cost data in process and risk management. On the other hand, the 1998 survey respondents were chosen to be representative of a variety of industries, regardless of sophistication or maturity. Thus, the 1995 survey respondents were more likely than those in the 1998 survey to be using the defect data in sophisticated. In fact, 9 of the 21 participants in the 1995 survey were either CMM level 2 or level 3. Of the companies participating in the 1998 survey, only one had a formal external assessment of the quality or maturity of its software development practices, although a second was in the process of such an assessment.

It is relatively safe to conclude that companies that track defect data fall into three camps. The first group, whom we call “firefighters,” collects defect information strictly for day-to-day management. Their defect tracking systems give the managers a way to assign defects to certain developers to be fixed. At the same time, the systems advise the tester when the defect has been fixed so the tester can verify the fix. They provide a mechanism for developers to query a system to find what defects are assigned to them; project managers can also use the systems to check the load on each developer. Some simple analysis can be done, such as tracking how many new defect reports were filed in a given time period. As one responder to the 1998 survey stated, when the number of new defects found starts to level off, either the development phase is coming to an end or the test suite is losing its effectiveness.

Most companies in this first camp need to collect a good deal of data about each defect in order to achieve the above stated goals. The most important detail is a textual summary of the defect, including the steps required to reproduce the defect. The date that a defect was found, the tester reporting the defect, the date the defect was repaired, and the developer responsible for the repair are all usually tracked. Most companies also track test environment, including what hardware and operating system the

test machine was running when the defect occurred. Error messages produced by the software or operating system are often logged, and in some companies, an identifier of which test case produced the defect is tracked. In some cases, defects are logged with a priority and/or a severity, which indicates either the tester's (or manager's) input on how to prioritize the repair of defects. Development groups frequently try to narrow down the location of the defect, either by tracking a functional area of the program, a module or a source file, depending on the group. All of these data can be used to classify defects. For example, all defects found during a certain phase would be one defect class. However, defect analysis is time-consuming and often suffers on resource-constrained projects.

The second group of companies takes the above one step further. These companies use the data in what Robert Grady terms a "reactive" fashion [Grady96]. The projects in these companies use the defect data locally within their project. Defect data are often not shared among projects. The primary use of the defect data in reactive organizations is to improve local situations [Grady96]. While the data gathered may identify "hot spots" in the current artifact or process, the emphasis is placed on fixing the problems at hand, rather than trying to prevent similar problems in the future. A typical "reactive" process would be similar to the process described in [OstrandWeyuker84] above, for example:

1. Researchers examine error data from a completed project or projects to develop a fault classification scheme.
2. At the end of the next project, researchers examine all problem (fault) reports and classify the faults using the scheme developed in step 1.
3. Researchers analyze data to try to determine if a process is in need of change.
4. After a project is developed with a process change, compare the fault data for this project to the data gathered in step 3 to measure improvement.

An advantage to having a company-wide metrics program such as the one at Hewlett-Packard is that teams have the ability to compare all of the above measures for the current project to the norm for the company at large. This helps teams identify where their project is deviating from the norm, which may be a signal that an area of the process or product needs more attention. In this situation, the above 4-step process could be amended to classify multiple projects in step 2 and then compare the data to a baseline in step 3. In practice, a "reactive" organization may begin with the original 4-step process listed above, and then as more data are gathered, shift gradually to the more evolved situation where comparisons to a baseline are used.

The third group of companies act in what Grady terms a "proactive" fashion, trying to analyze the defects encountered for the purpose of preventing those types of defects from occurring in the future [Grady96]. The key to becoming a proactive organization is some sort of focus on the root cause of the defect. Grady describes "root cause analysis" and describes a meeting format in [Grady96] which is very similar to the Defect Prevention work at IBM. In a proactive organization, Grady states, defect data must be shared among projects so that company-wide trends can be identified. The knowledge gained from the analysis of errors, faults, and failures must be used to improve processes and identify what types of training are necessary for employees. Finally, there must be some form of continuous feedback loop so that the organization can continually improve their development processes.

III. Support Mechanisms

The preceding section of this report examined the defect analysis and classification efforts as published in research journals. Two questions remain to be addressed to those who are looking to begin defect analysis efforts. The first addresses the repeatability (and thus the feasibility) of code defect classifications. If two engineers classify the same defect differently, then the defect classification data will be of questionable validity and conclusions reached from the data may not be significant or effective. The second question involves implementation. How should an organization with little or no formal defect analysis or classification efforts go about implementing these efforts?

A. Repeatability

The first question on repeatability has been addressed in an empirical study conducted by Khaled El Emam and Isabella Wieczorek of the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern, Germany [ElEmam98]. The researchers examined data from several inspections during a project conducted within a German company. All inspections involved two reviewers. The reviewers classified each defect using a scheme very similar to the original Orthogonal Defect Classification scheme presented in the previous section. While only approximately 25% of defects found in a given inspection were discovered by both reviewers, in the situation where both reviewers discovered the same defect, they classified it the same way a statistically significant portion of the time. This study examines only a single scheme in a single environment, but it gives an independent confirmation of the results experienced by the practitioners of the methods discussed in section II.

B. Goal Setting

The first step in implementing a defect analysis program must be the establishment of well-defined goals. Goals are necessary so that analysis efforts can be focused and coordinated. One example of a goal could be attempting to characterize or understand some attribute of the development environment. Another could be analyzing the impact of a specific process change in the environment. When defining goals, it would be wise to consider the maturity of the organization as a whole on a scale such as the CMM. Additionally, one should consider the organization's "defect" maturity ("firefighter," "reactive," or "proactive") as described above. The state of the organization on scales such as these can help the organization to generate goals that are reasonably achievable. On the other hand, if the organization has a goal of improving CMM maturity, the goals could be set high enough to allow progress in this area.

[BasiliWeiss84] and [BasiliRombach87] discuss a specific process for setting goals. These researchers identify setting goals as the first step of the Goal-Question-Metric paradigm (GQM). To them, a goal has four components: a Purpose, an Issue, an Object (or Process) and a Viewpoint. The Purpose is a verb, such as "To Characterize" or "To Improve." The Issue is the object of the verb used as the purpose, such as "Defect Type Distribution" or "Defect Repair Effort." The Object or Process describes the artifact being examined, such as "Modified Code Modules" or "Software Design Process." Lastly, the Viewpoint encompasses the user of the data as well as the scope of the analysis. For example, the Viewpoint could be the development manager, where the goal would involve giving feedback on attributes of a specific project. Alternatively, the viewpoint could represent a researcher in the organization trying to compare processes from multiple divisions. Similarly, the viewpoint of a high-level manager might reflect a desire to evaluate performance of all the projects in a department. The

four attributes of a goal can be combined into a single sentence that concisely captures the goal. For example, a goal could be "To reduce defect repair effort in modified code modules from the point of view of the development manager." The possibilities for combinations of these attributes are limited only by one's imagination.

Robert Grady in [Grady87] also lists establishing goals as the first step to any successful software measurement program. He points out that a program's objectives help determine how the program will be implemented, and in [Grady96] he adds that the goals allow the organization to prioritize efforts. This prioritization allows the organization to focus resources in a manner that will maximize the likelihood of the program's success. Be forewarned that if goals are not well-specified, "one runs the risk of collecting unrelated, meaningless data," or worse "data in which either incomplete patterns or no patterns are discernible" [BasiliWeiss84].

Goals are also important from a business perspective. In order for a defect analysis program to be successful, management and development teams must agree on the goals. To gain management support, one may try to show the cost advantages if the goals are achieved. The defect prevention research team from section II.E above attempted to calculate return on investment by estimating the number of faults and errors prevented and the effort saved by not having to find and fix them. Often a specific numerical return on investment will be difficult if not impossible to determine at the start of a new program, but management will likely want to feel that some tangible benefit will be gained before agreeing to commit the necessary resources. If management does not agree to protect the defect analysis efforts from the seemingly inevitable cuts in resources as schedules get tight, then it will be very difficult for the program to succeed. To help avoid this situation, [Grady87] suggests that the responsibilities of each member of the organization be carefully enumerated before proceeding. In this way, the effort required can be estimated and factored into schedules. Finally, if the goals involve changing of development processes, which they often do in mature organizations, it is imperative that management and development agree on the method for implementing process changes.

Once the goals have been established, and management has agreed to support the efforts to reach them, the next step is to divide the goal into its major subcomponents [BasiliRombach87]. This decomposition can be accomplished by generating a list of "Questions of Interest," the second level of the GQM hierarchy [BasiliWeiss84]. The questions describe how the goal is going to be achieved. Consider again the sample goal described earlier, "To reduce defect repair effort in modified code modules from the point of view of the development manager." One question that would need to be answered is "What is the current effort spent repairing defects in modified modules?" It is possible for one question to address multiple goals. For example, the above question could also address the goal "To characterize the costs and benefits of the code reuse process from the point of view of the high-level manager."

Questions of interest connect the goals to the data needed to satisfy the goals. At the same time, they allow an organization to refine the goals to a finer level of detail. If questions cannot be generated for a specific goal, then perhaps the goal needs to be defined more clearly. As stated in [BasiliWeiss84], "goals are not well defined if questions of interest are not or cannot be formulated." Omitting this step can be costly. If the questions of interest are missing, then there is too little guidance for what data must be collected. This may lead to inaccurate or incomplete data which may prevent the goals from being adequately satisfied [BasiliWeiss84].

After the goals have been broken down into their component questions, the next step is to determine what data will be collected. These data, also called metrics, form the third level in the GQM hierarchy. The metrics can be objective or subjective. The difference between the two is that subjective metrics depend on the point of view of the person generating the metrics, but objective metrics do not [BasiliCaldieraRombach94]. The data gathered must address the questions of interest generated in the previous step. Just as one question may help address more than one goal, one metric may partially answer more than one question. One of the advantages of establishing goals before choosing metrics can be seen here: since each metric must be traceable to at least one goal through at least one question, there is no collection of needless data. Figure 7 shows graphically the relationships between Goals, Questions of Interest, and Metrics in the GQM hierarchy, figure 8 lists a detailed example of a Goal and all the Questions of Interest and Metrics needed to meet that goal.

The survey of literature presented in section II gives some insight as to what metrics should be collected. The location of the defect both generally (e.g. requirements, design, code) and specifically (a particular module) can be useful for determining where processes can use improvement. The method used to detect the defect can help evaluate the success of various defect-detection methods. The effort required to repair a defect gives an objective measure of the severity of the defect, although this one measure alone does not give a complete picture of severity. The above metrics are all objective and simple to collect.

There are subjective metrics which would be useful to collect as well. Several of the classification efforts described in section II involved classifying defects by type. The type scheme described in the Orthogonal Defect Classification work in section II.E attempts to minimize the possibility of misclassifying defects. The classification scheme for requirements faults presented in [BasiliWeiss81] in section II.A (Omission, Incorrect Fact, Ambiguity, etc.) is still being used by researchers at the University of Maryland today. While these schemes may not fit perfectly into other environments, they make excellent starting points for practitioners interested in developing their own schemes.

Once the metrics have been enumerated, the next steps include to preparing the personnel for collecting the data and then actually collecting the data. Before data collection, the development staff needs training in the collection processes. If the staff does not fully understand the process, the data collected may not be accurate, and the defect analysis effort will likely fail. It is wise to share the goals of the defect analysis effort with the staff, so that they know why they are collecting this data. If they understand they goals, the staff may be more motivated and desire to accomplish the task successfully.

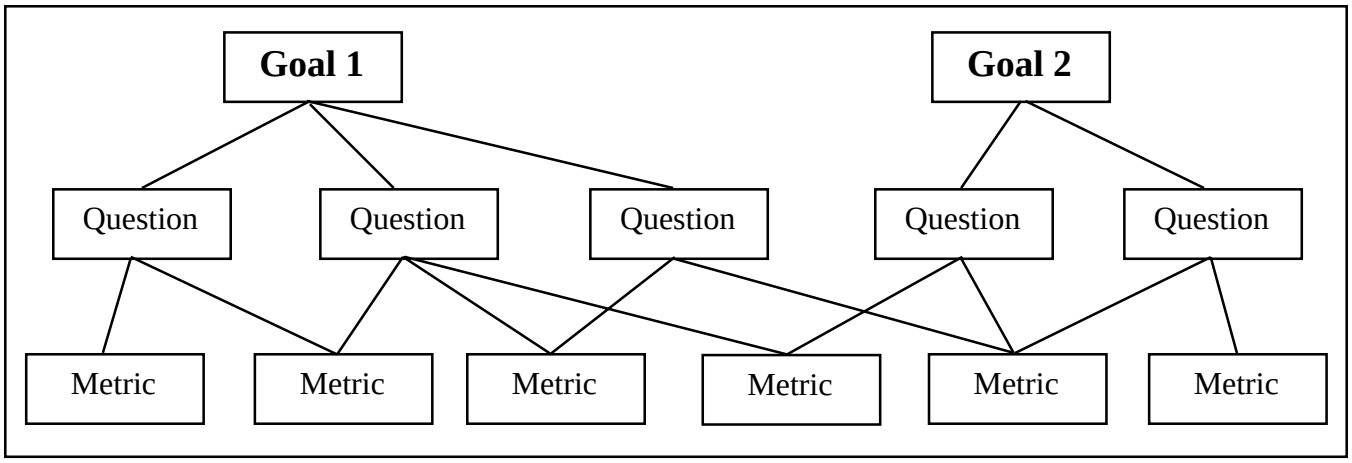


Fig 7. GQM model hierarchical structure [BasiliCaldieraRombach94]

Goal	Purpose Issue Object (process) Viewpoint	Improve the timeliness of change request processing from the project manager's viewpoint
Question	Q1	What is the current request processing speed?
Metrics	M1 M2 M3	Average Cycle Time Standard deviation % cases outside of the upper limit
Question	Q2	Is the (documented) change process actually performed?
Metrics	M4 M5	Subjective rating by project manager % of exceptions identified during reviews
Question	Q3	What is the deviation of the actual change request processing time from the estimated time?
Metrics	M6 M7	$\frac{\text{Current average Cycle Time} - \text{Estimated Average Cycle Time}}{\text{Current average Cycle Time}} * 100$ Subjective evaluation by Project manager
Question	Q4	Is the performance of the process improving?
Metrics	M8	$\frac{\text{Current average cycle time}}{\text{Baseline average cycle time}} * 100$
Question	Q5	Is the current performance satisfactory from the point of view of the project manager
Metrics	M7	Subjective evaluation by Project manager
Question	Q6	Is the performance visibly improving?
Metrics	M8	$\frac{\text{Current average cycle time}}{\text{Baseline average cycle time}} * 100$

Fig 8. Example GQM Model [BasiliCaldieraRombach94]

C. Tools and Procedures

Tools are useful for facilitating the data collection process. For example, in the past, defect data were often collected on paper forms, similar to the Hewlett-Packard Defect Classification Worksheet shown in section II.C above. However, many forms are available now in electronic versions. Most of the defect-related tools are simply configurable interfaces to databases, where an administrator can configure the fields that must be filled out for each defect reported. Some tools also have the feature of enforcing predetermined workflow restrictions. For example, the tool could be programmed with a list of developers, a list of managers and a list of testers. Once a tester has discovered a defect, the only person who can alter the state of the defect is a manager, who selects the developer to which the defect is assigned. Once the developer has indicated that the defect has been repaired, only a tester can indicate that the repair was successful.

There are benefits to enforcing workflow restrictions, but there are also drawbacks if the restrictions are too rigid. For example, one company that responded to the 1998 survey (discussed in section II.F above) originally chose to enforce the restriction that only the tester who originally discovered the defect could close out the defect and mark the repair successful. The thinking was that the original tester would be most familiar with the defect and thus in the best position to make the judgment on the quality of the repair. However, if the tester were on vacation or left the company, it would not be possible to mark the defects as repaired without getting a database administrator to override the workflow restriction. This situation became too burdensome, and eventually the company changed defect tracking tools. A second company that also responded to the 1998 survey chose not to enforce any workflow restrictions in their defect tracking tool. The tool allows any engineer regardless of position to make any addition or change to any defect in the system at any time. However, the system also keeps a detailed log of what changes were made, when they were made and by whom, so no information is ever lost.

After the data are collected, they must be validated. As previously mentioned in the discussion of the efforts at the Software Engineering Laboratory at NASA-Goddard (section II.A above), a separate team can inspect the defect data for inconsistencies and contradictions. In [Weiss81], dedicated error analysts would interview developers whenever the analyst did not understand something on the error report form, and [WeissBasili85] mentions interviewing an engineer whenever a possible discrepancy arises with some data that the engineer collected. Additionally, more general interviews can solicit candid comments on how closely the engineers conformed to the specified process. If process conformance is a problem, perhaps the data collection process needs to be altered or maybe the engineers need more training. It is also important to determine whether or not the data collection efforts are complete, in that every defect is properly entered and tracked in the system [BasiliWeiss84]. Without validation, as many as half of the defect forms collected may have at least one inaccuracy [BasiliWeiss84]. It is impossible to make correct inferences from inaccurate data, so it is essential to devote some resources to verifying the validity of the data.

The next step involves analyzing the data. This step most reflects the maturity of the organization, and also evolves the most over time. Data analysis will likely be very simple in an organization just moving from the “firefighter” stage to the “reactive” stage, or an organization starting at CMM level 1 and slowly making efforts to improve maturity. As more and more data are collected, a repository of data can be built, and a baseline can be generated. At this point, data analysis may involve statistical analysis, root cause analysis, or any of the efforts described in section II above.

After the analysis itself is conducted, the final step involves feedback, both to the individuals who participated in the project and the organization as a whole. The results of the analysis may suggest changes that may reduce defect insertion or enhance defect prevention, detection and correction. Successful process change, as well as evidence of its effect, should be publicized so that participants know that their efforts have made a difference [Grady87]. As developers learn that they can actually affect decisions on process improvements, morale will improve and developers will be more likely to make suggestions and participate in defect analysis and classification efforts [Mays90]. As an organization evolves, the feedback may also include a more formalized component, such as a repository of experience as discussed in the defect prevention work (section II.E above). This common body of knowledge allows an entire organization to learn from the experiences of each team, another key component to improving the maturity of the environment.

D. Conclusion

In conclusion, the following process should be followed by an organization seeking to implement or improve upon a defect classification and analysis effort:

1. Establish well-defined goals.
2. Get management support for the effort and agreement on the goals and how to implement development process changes.
3. Divide the goal into subcomponents by generating "Questions of Interest."
4. Determine the metrics to collect whose values will help developers and managers answer the "Questions of Interest."
5. Train personnel in defect data collection methods and tools, where tools will help to support data collection, tracking and analysis.
6. Collect the data.
7. Validate the data.
8. Analyze the data.
9. Publish results and give feedback. Eventually this feedback will flow into step 1, helping to generate new goals so the organization can grow and mature. Likewise, the data gathered in step 6 will eventually populate a large repository from which a baseline can be generated, thereby improving the analysis in step 8.

What can an organization expect from following the nine-step process described above? The organization will gain understanding about both the products being developed and the development processes. At first, the development manager will only be able to look at a small set of defect data and use logic and gut feeling to try to identify areas that need improvement, but as time goes by the organization will be able to build a baseline which will allow the managers to run statistical analyses to analyze the product and processes. At this point, the level of understanding will allow the development teams to focus their efforts on improving processes. The organization will be able to better understand the strengths and weaknesses of their own environment and will be able to have a concrete measure of how much a process change improves software quality. Finally, the engineers will know from the feedback cycle that their opinions matter and that they can view processes as a way of improving quality rather than simply as constraints that must be lived with. The end result is a mature organization which understands its environment, learns from its experiences, and improves its processes in a manner consistent with the goals and needs specific to that organization.

IV. Acknowledgments

We would like to thank Shari Lawrence Pfleeger and Forrest Shull for their valuable comments on this report.

V. References

[BasiliCaldieraRombach94] Basili, Victor R., Gianluigi Caldiera, and H. Dieter Rombach (1994). "Goal Question Metric Paradigm." Reprinted from *Encyclopedia of Software Engineering – 2 Volume Set*. John Wiley & Sons, Inc.

[BasiliPerricone84] Basili, Victor R., and Barry T. Perricone (1984). "Software Errors and Complexity: An Empirical Investigation." *Communications of the ACM*, 27(1) (January):42-52.

[BasiliRombach87] Basili, Victor R., and H. Dieter Rombach (1987). "Tailoring the Software Process to Project Goals and Environments." *Proceedings: Ninth International conference on Software Engineering*, 345-357.

[BasiliWeiss81] Basili, Victor R., and David M. Weiss (1981). "Evaluation of a Software Requirements Document by Analysis of Change Data." *Proceedings: Fifth International Conference on Software Engineering*, 314-323.

[BasiliWeiss84] Basili, Victor R., and David M. Weiss (1984). "A Methodology for Collecting Valid Software Engineering Data." *IEEE Transactions on Software Engineering*, SE-10(6) (November):728-738.

[BasiliZelkowitz78] Basili, Victor R., and M. Zelkowitz (1978). "Analyzing medium Scale Software Developments," *Proceedings: Third International Conference on Software Engineering*, 116-123.

[Chillarege92] Chillarege, Ram, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong (1992). "Orthogonal Defect Classification: A concept for In-Process Measurements." *IEEE Transactions on Software Engineering*, 20(6):476-493.

[ElEmam98] El Emam, Khaled, and Isabella Wieczorek (1998), "The Repeatability of Code Defect Classifications." Technical Report. International Software Engineering Research Network, ISERN-98-09.

[Endres75] Endres, Albert. (1975). "An Analysis of Errors and Their Causes in System Programs." *Proceedings: International Conference on Reliable Software*, 327-336.

[GradyCaswell87] Grady, Robert B., and Deborah Caswell (1987). *Software Metrics: Establishing a Company-Wide Metrics Program*. Englewood Cliffs, NJ: Prentice-Hall.

[Grady92] Grady, Robert B. (1992). *Practical Software Metrics for Project Management and Process Improvement.* Englewood Cliffs, NJ: Prentice-Hall.

[Grady96] Grady, Robert B. (1996). "Software Failure Analysis for High-Return Process Improvement Decisions." *Hewlett-Packard Journal*, 47(4) (August).

[IEEE83] IEEE (1983). *IEEE Standard 729: Glossary of Software Engineering Terminology.* Los Alamitos, CA: IEEE Computer Society Press.

[Mays90] Mays, R.G., C. L. Jones, G. J. Holloway, and D. P. Studinski (1990). "Experiences with Defect Prevention." *IBM Systems Journal*, 29 (1) (January):4-32.

[OstrandWeyuker84] Ostrand, Thomas S., and Elaine Weyuker (1984). "Collecting and Categorizing Software Error Data in an Industrial Environment." *The Journal of Systems and Software*, 4:289-300.

[Pfleeger98] Pfleeger, Shari Lawrence (1998). *Software Engineering: Theory and Practice*. Upper Saddle River, NJ: Prentice-Hall.

[Shull98] Shull, Forrest (1998). Developing Techniques for Using Software Documents: A Series of Empirical Studies. PhD Thesis, Computer Science Department, University of Maryland.

[Weiss78] Weiss, David M. (1978) *"Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility."* Technical Report 8268. Naval Research Laboratory, Washington DC.

[WeissBasili85] Weiss, David M., and Victor R. Basili (1985). "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory." *IEEE Transactions on Software Engineering*, SE-11(2) (February):157-168.