SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(10), 1095–1105 (OCTOBER 1993)

An Application of Causal Analysis to the Software Modification Process

JAMES S. COLLOFELLO
Computer Science Department, Arizona State University, Tempe, AZ 85287, U.S.A.

AND

BAKUL P. GOSALIA
AG Communication Systems Corporation, Phoenix, AZ 85027, U.S.A.

SUMMARY

The development of high quality large-scale software systems within schedule and budget constraints is a formidable software engineering challenge. The modification of these systems to incorporate new and changing capabilities poses an even greater challenge. This modification activity must be performed without adversely affecting the quality of the existing system. Unfortunately, this objective is rarely met. Software modifications often introduce undesirable side-effects, leading to reduced quality.

In this paper, the software modification process for a large, evolving real-time system is analysed using causal analysis. Causal analysis is a process for achieving quality improvements via fault prevention. The fault prevention stems from a careful analysis of faults in search of their causes. This paper reports our use of causal analysis on several significant modification activities resulting in about two hundred defects. Recommendations for improved software modification and quality assurance processes based on our findings are also presented.

KEY WORDS: Causal analysis Software maintenance

BACKGROUND

The traditional approach to developing a high quality software product consists of applying a development methodology with a heavy emphasis on fault detection. These fault detection processes consist of walkthroughs, inspection and various levels of testing. A more effective approach to developing a high quality product is an emphasis on fault prevention. An effective fault prevention approach which is gaining popularity is causal analysis.[1–4] Causal analysis consists of collecting and analyzing software fault data in order to identify their causes. Once the causes are identified, process improvements can be made to prevent future occurrences of the faults.

The paper by Jones[3] presents a programming process methodology for using causal analysis and feedback as a means for achieving quality improvement, and ultimately fault prevention, at IBM. The methodology emphasizes effective use of the fault data to prevent the recurrence of faults.

The fault prevention methodology is based on the following three concepts:

0038–0644/93/101095–11$10.50 Received 6 April 1993
Revised 6 April 1993

1. Designers should evaluate their own faults.
2. Causal analysis should be part of the software development process.
3. Feedback should be part of the process.

An overview of the causal analysis process proposed by Jones is shown in Figure 1. The first activity consists of a kickoff meeting with the following purposes:

1. Review input fault data.
2. Review the methodology guidelines.
3. Review appropriate checklists.
4. Identify the team goals.

Next, the development of the work products occurs, using the feedback from the kickoff meeting to prevent the creation of errors.

The work products are then validated and reworked as necessary.

A causal analysis activity is then performed, beginning with an analysis of the faults performed by the owner of the faults. This involves analyzing each fault to determine

1. The fault type or category
2. The phase that the fault was found
3. The phase that the fault was created
4. The cause(s) of the fault
5. The solution(s) to prevent the fault from occurring in the future.

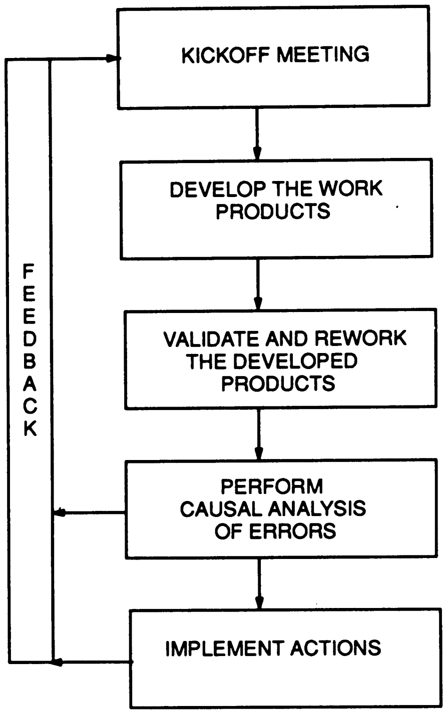
KICKOFF MEETING
DEVELOP THE WORK PRODUCTS
VALIDATE AND REWORK THE DEVELOPED PRODUCTS
PERFORM CAUSAL ANALYSIS OF ERRORS
IMPLEMENT ACTIONS
FEEDBACK

Figure 1. Causal analysis process described by Jones

This information is recorded on a causal analysis form which is used to enter the data into a database. A causal analysis team meets to analyze the data in the database. In addition, the group may at times need to consult with other people (designers, testers, etc.) outside of the team to complete the analysis.

The causal analysis team is responsible for identifying the major problem areas by looking at the fault data as a whole instead of one particular fault at a time. The team uses a problem-solving process to analyze the data, determine the problem area(s) to work on, determine the root causes of problem area(s), and develop recommendations and implementation plans to prevent the problem type(s) from occurring in the future.

These recommendations are then submitted to an action team, which has the following responsibilities:

1. Evaluation and prioritization of recommendations.
2. Implementation of recommendations.
3. Dissemination of feedback.

The action team meets periodically (e.g. once a month) to review any new implementation plans received from the causal analysis teams and to check the status of the previous implementation plans and action items. The status of the action items is kept in the causal analysis database and monitored by the action team.

To date, most of the effort reported in causal analysis activities has been focused on development processes. This is unfortunate, considering the high percentage of effort consumed by software modification processes. Software modifications occur in order to add new capabilities or modify existing capabilities of a system. Modifying a large, complex system is a time-consuming and error-prone activity. This task becomes more difficult over time as systems grow and their structure deteriorates.[5]

Thus, more effort must be expended in performing causal analysis activities for design maintenance tasks. This paper describes such an effort. In the remainder of this paper, our approach to causal analysis for design maintenance tasks is described, as well as the application of this approach to a large project. Our results and recommendations for preventing faults during modification processes are also presented.

CAUSAL ANALYSIS APPROACH

Our application of causal analysis to the software modification process is shown in Figure 2. Each of the causal analysis steps in Figure 2 is described in one of the following subsections.

Obtaining problem reports

The first activity to be performed is collecting problem reports resulting from the modification processes undergoing causal analysis scrutiny. These problem reports may be generated internally via testing or externally by the customer. A window of time must be chosen during which the problem reports are collected. Ideally, the window of time for obtaining problem reports should begin after the code is modified and end some period of time after the code was delivered.

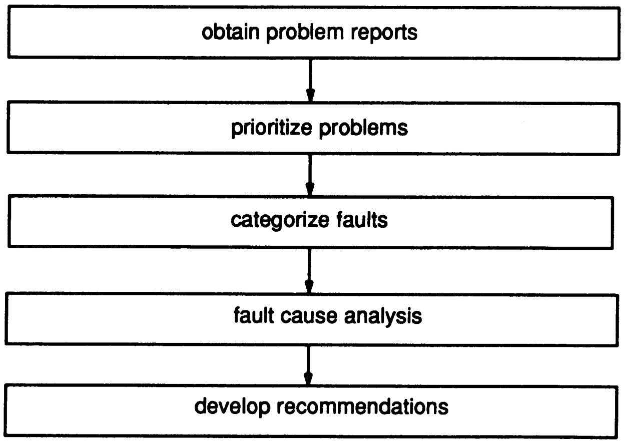

Figure 2. Causal analysis approach as applied to the software modification process

Prioritizing problems

The next step in our causal analysis approach is prioritizing the faults. Fault prioritization enables an analysis of those causes which contribute to high priority problems. Our prioritization consists of three categories:

1. Critical
2. Major
3. Minor

Critical faults impair functionality and prohibit further testing. Major faults partially impair functionality. Minor faults do not affect normal system operation.

Categorizing errors

After fault prioritization, faults are categorized. The primary objective of this categorization is to facilitate the link between faults and their causes. Numerous error categorization schemes have been proposed over the years.[6–9] The existing schemes are, however, directed towards understanding and improving the development of new software. They do not address the kinds of errors that are unique to maintenance activities. Thus, we found it necessary to expand upon existing categorization schemes and develop new error categories directed towards maintenance activities. Our categories were formulated based on our experience of analyzing hundreds of maintenance errors.

Although categorization is not necessary in performing a causal analysis, we found the categorization useful in identifying fault causes. The fault categorization also provides useful information when determining the cost-effectiveness of eliminating fault causes versus attempting to detect the faults if they occur.

The fault categories are described below:

Design faults

This category reflects software faults caused by improper translation of requirements into design during modification. The design at all levels of the program and data structure is included. The following are examples of typical design faults:

1. *Logic faults*. Logic faults include sequence faults (improper order of processing steps), insufficient branch conditions, incorrect branch conditions, incorrectly nested loops, infinite loops, incorrectly nested if statements, missing validity checks etc.
2. *Computation faults*. Computation faults pertain to inaccuracies and mistakes in the implementation of addition, subtraction, multiplication, and division operations. Computational faults also include incorrect equations or grouping of parenthesis, mixing data of different unit values, or performing correct computations on the wrong data.
3. *Missing exception handling*. These faults include failure to handle unique conditions or cases even when the conditions or cases were specified in the requirements documents.
4. *Timing faults*. This type of fault reflects incorrect design of timing critical software. For example, in a multitasking system, a designer interpreted the executive command ‘suspend’ to mean that all processing is halted for a specified amount of time, when in fact the command suspended all tasks scheduled to be invoked in the same specified amount of time.
5. *Data handling faults*. These faults include failure to initialize the data before use, improper use or designation of fixes, mixing up the names of two or more data items, and improper control of external file devices.
6. *Faults in I/O concepts*. These faults include input from or output to the wrong file or on-line device, defining too much data for a record size, formatting faults, and incorrect input–output protocols with other communication devices.
7. *Data definition faults*. This category reflects incorrect design of the data structures to be used in the module or the entire software (incorrectly defined global data structures). This category also reflects incorrect scoping in data structures and incorrect data abstractions.

Incompatible interface

Incompatible interface faults occur when the interfaces between two or more different types of modified components are not compatible. The following are examples of incompatible interfaces:

1. Different parameters passed to the procedures/functions than expected.
2. Different operations performed by the called modules, functions, or procedures than expected by the calling modules, procedures or functions. For example, assume a module A was called by another module B to perform fault checking. However, as a design change, the fault checking code was moved to another software module C.
3. Code and database mismatched. For example, suppose that device handler code tries to access data which is not defined in the database.
4. Executive routines and other routines mismatched. For example, consider the

situation where a task needs to remain in a no interrupt condition for a longer time than allowed by the operating system routines.
5. Software and hardware mismatched.
6. Software and firmware mismatched. For example, the parameters passed by the firmware may not match with the modified software.

Incorrect code synchronization from parallel projects

For a large evolving system, the software development cycles of multiple releases may overlap, as shown in Figure 3.

This category of fault occurs when changes from project A, the previous project, are incorrectly carried into project B, the current project.

Incorrect object patch carryover

In a large maintenance effort, object patches are sometimes needed. These object patches may remain for several revisions of the system. Errors committed while modifying modules with object patches fall into this category.

System resource exhaustion

This fault occurs when the system resources (such as memory and real time) become insufficient.

Examples of faults in this class include: exhaustion of available stack space, the indiscriminate use of special registers and the use of non-existent stacks, real-time clocks, and other architectural features.

Determining fault causes

The most critical step in performing causal analysis is determining the cause of each fault. This task can best be performed by the programmer who introduced the fault. To facilitate this identification of fault causes, the software designers who introduced the faults should be interviewed. The interviews should be informal and performed very carefully so as to not make designers defensive. The purpose of the analysis for defect prevention must be made clear prior to the interview. Based on the information acquired from the designer who introduced the fault, a causal category is selected for the error. Our approach to causal analysis identifies seven major categories of causes that lead to software modification faults.

Our causal categories differ from others in the literature[5,9] which are directed

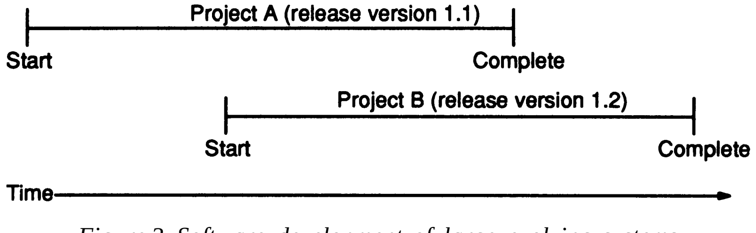
Project A (release version 1.1)
Start
Complete
Project B (release version 1.2)
Start
Complete
Time

Figure 3. Software development of large evolving systems

towards finding the causes of errors made during new software development. These existing causal categorization schemes do not address the unique causes of errors made during software maintenance.

Our causal categories are not exclusive and, thus, a fault might have several causes. The causal categories were formulated in a manner that would support process improvements. The goal of determining the causal category for each fault is to collect the data necessary to determine which causal categories are leading to the most problems. This provides a means for evaluating the cost effectiveness of implementing recommendations to eliminate or reduce fault causes.

Our causal categories customized to modification activities are described below:

System knowledge/experience

This causal category reflects the lack of software design knowledge about the product or the modification process. Some examples include:

1. *Misunderstanding of the existing design*. For example, the designer did not have sufficient understanding of the existing design of the database.
2. *Misunderstanding of the modification process*. The designer was not aware of the process followed in the project. For example, the designer was not aware of the configuration control process.
3. *Inadequate understanding of the programming environment*. The designer did not have adequate understanding of the programming environment. The programming environment includes personnel, machinery, management, development tools, and any other factors which affect the developer's ability to modify the system.
4. *Inadequate understanding of the customer's requirements*. The designer did not thoroughly understand the requirements of the customer. For example, the designer may not understand a feature specified in the requirements documents.

Communication

This causal category reflects communication problems concerning modifications. It identifies those causes which are not attributed to a lack of knowledge or experience, but instead to incorrect or incomplete communication. Communication problems are also caused due to confusion between team members regarding responsibilities or decisions.

Some examples of communication problems are:

1. Failure of a design group to communicate a last-minute modification.
2. Failure to write a fault-handling routine for a function because each team member thought someone else was supposed to do it.

Software impacts

This category reflects failure of a software designer to consider all possible impacts of a software modification.

An example is omission of fault recovery code after the addition of a new piece of hardware.

Methods/standards

This category reflects methods and/or standards violations. It also includes limitations to existing methods or standards which contributed to the faults.

An example may be a missed code review prior to testing.

Feature deployment

This category reflects problems caused by the inability of software, hardware, firmware or database components to integrate at the same time. It is characterized by a lack of contingency plans to prevent problems caused by missing components.

Supporting tools

This category reflects problems with supporting tools which introduce faults. An example is an incorrect compiler which introduces faults in object code.

Human error

This causal category reflects human errors made during the modification process which are not attributable to other sources. The designer knew what to do and understood the item thoroughly but simply made a mistake.

An example may be that a designer forgets to add code he intended for a software load.

Developing recommendations

The last step in our causal analysis process is developing a set of recommendations to prevent faults from reoccurring. These recommendations must be based on the data collected during the causal analysis study and customized to the development environment.

APPLICATION OF THE CAUSAL ANALYSIS APPROACH

In an effort to learn more about the faults generated during software modifications, we applied our causal analysis process to the creation of a new version of a large telephony system. This new version contained new features as well as fixes to problems in the old release. The size of the system was approximately 2·9 million lines of code. The version analyzed was developed from the previous version by gradual modifications to the software, firmware, and hardware. About 1 million lines of code were modified as follows: 85 per cent of the code was added, 10 per cent of the code was changed, and 5 per cent of the code was deleted. Each of these gradual modifications corresponded to a software increment. The increments were added to the system one at a time and tested as shown in Figure 4. The new version we analyzed consisted of 11 increments. The experience levels of the designers involved in modifying the software varied widely from two years to as high as 30 years.

The documentation used and created in the process of generating the new version was typical of that in most software development organizations. Both high level and

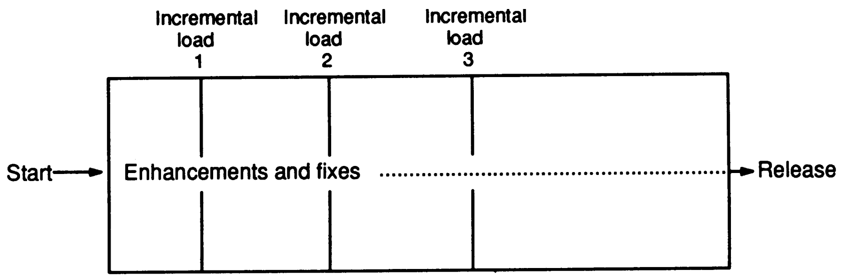

Figure 4. Gradual building of a software version release

detailed design documents were created for the new features being added. The contents of these documents included details about the impacts of the new feature on the existing modules, hardware, database, firmware, etc. These documents were then reviewed by designers representing the affected areas. Documentation was also generated and reviewed for fixes added to the new version.

Although it would have been ideal to collect and analyze problem reports for the entire modification process, our study was limited to analyzing defects detected during the first phase of integration testing for each of the incremental loads. This phase verifies the basic stability and functionality of the system prior to testing the operation of new features, functions, and fixes. It is, thus, the most likely phase to detect faults introduced by the modification process. Prior to our data collection, both code reviews and module testing were generally applied. Thus, faults detected by these processes were not reflected in our results. In addition, faults detected by later phases of testing, such as system testing and stress testing were also not reflected in our results. Our results, therefore, are limited to the kinds of faults normally detected during integration testing. In our study, this amounted to more than 200 faults. All 200 faults were analyzed and categorized by this paper's authors to ensure reliable results.

RESULTS OF THE CAUSAL ANALYSIS APPLICATION

Figure 5 presents an overall distribution of fault categories across all the problems.

The large number of design and interface faults demonstrates that the modification

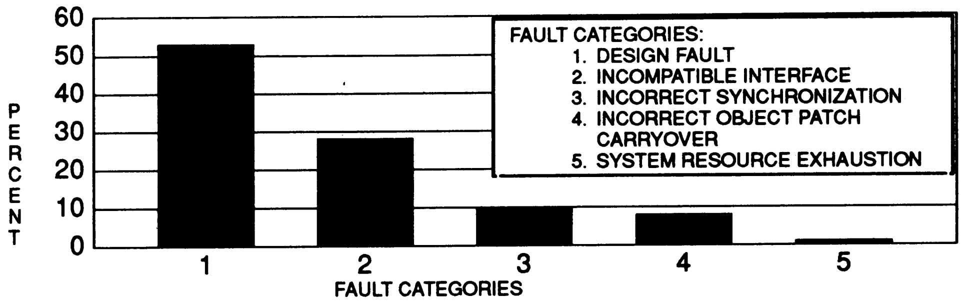

Figure 5. Fault category distribution of all the problems

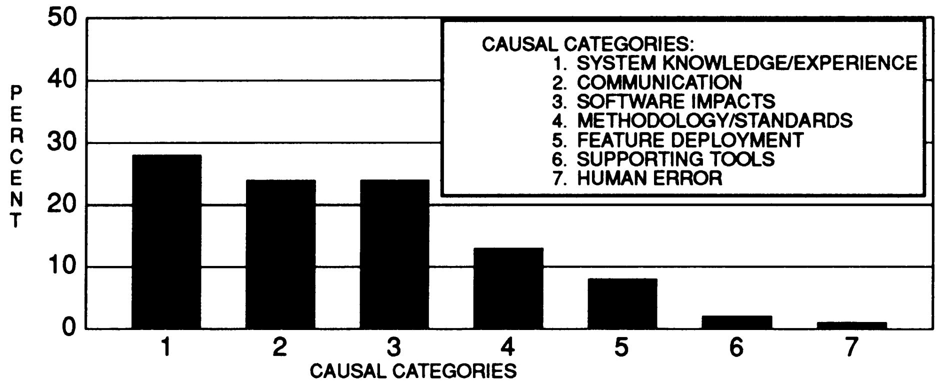

Figure 6. Causal categories distribution of all the problems

of an existing design in order to satisfy new requirements is a very error-prone process.

Figure 6 presents the distribution of causal categories for all the problems.

The causal analysis data suggests that almost 80 per cent of all faults are caused by insufficient knowledge/experience, communication problems or a failure to consider all software impacts. A further analysis of the faults caused by insufficient knowledge/experience indicated that the majority of those faults were caused by designers with less than five years of experience.

These results clearly suggest that one significant approach to improving the quality of maintenance activities is to increase the experience level of those performing design maintenance tasks. This can be accomplished in many ways. One obvious approach is motivating designers to continue in their work assignments. Another approach is to improve design maintenance training and mentoring programs.

The results obtained from the causal analysis also suggest that the design maintenance process can be greatly improved via better communication. This might be accomplished with more disciplined maintenance methodologies, more thorough documentation and diligent reviews. Extreme care must be taken to ensure that information regarding potential software impacts as a result of modification activities are communicated.

The results from our causal analysis activity were integrated with those of other ongoing continuous process improvement approaches within our organization and applied to the development of a subsequent version of the software. The improvements mainly focused upon an enhanced review process, better training and better communication. The fault data from the subsequent version of the software release showed a substantial reduction in the number of faults.

CONCLUSIONS AND FUTURE RESEARCH

This paper has presented our results from applying causal analysis to several large modification activities. Our data suggests that almost 80 per cent of all faults created during modification activities are caused by insufficient knowledge/experience, communication problems or a failure to consider all software impacts.

The application of causal analysis to the software modification process has contributed to a substantial reduction of faults in our organization. Much additional research needs to be performed to assess whether the results of this study are representative of that experienced in other large evolving systems. Other research needs to explore the attributes of an 'experienced' designer. Specific design maintenance methodology improvements must also be formulated and evaluated in terms of their potential return on investment.

REFERENCES

1. J. L. Gale, J. R. Triso and C. A. Burchfield, 'Implementing the defect prevention process in the MVS interactive programming organization', *IBM Systems Journal*, **30**, (1), 33–43 (1990).
2. R. G. Mays, C. L. Jones, G. J. Holloway and D. P. Studinski, 'Experiences with defect prevention', *IBM Systems Journal*, **30**, (1), 4–32 (1990).
3. C. L. Jones, 'A process-integrated approach to defect prevention', *IBM Systems Journal,* **24**, (2), 50–164 (1985).
4. R. T. Philips, 'An approach to software causal analysis and defect extinction', *IEEE Globecom 1986,* **1**, (12), 412–416 (1986).
5. J. Collofello and J. Buck, 'Software quality assurance for maintenance', *IEEE Software*, No. 5, September 1987, pp. 46–51.
6. A. Endres, 'An analysis of errors and their causes in system programs', *IEEE Trans. Software Eng.*, **SE-1**, (2), 140–149 (1975).
7. B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 1983.
8. J. Collofello and L. Blumer, 'A proposed software error classification scheme', *Proceedings of the National Computer Conference*, 1985, pp. 537–545.
9. T. Nakajo and H. Kume, 'A case history of software error cause–effect relationships', *IEEE Trans. Software Eng.,* **17**, 830–837 (1990).