A proposed causative software error classification scheme

by JAMES S. COLLOFELLO
Arizona State University
Tempe, Arizona

and

L. BLUMER BALCOM
GTE Sylvania
Mountain View, California

ABSTRACT

Various tools, techniques, and methodologies have been developed by software engineers over the last 15 years. A goal of many of these approaches is to increase product reliability and reduce its cost by decreasing the number and severity of errors introduced by the software development process. The collection of software error data would appear to be a natural means for validation of these software engineering techniques. Yet, current software error collection efforts have had limited success in this area. A new causative software error classification scheme is introduced in this paper to refine these data collection efforts so that they can be better used in software engineering validation studies.
INTRODUCTION

The major expenses in most computer systems can be attributed to their software components. Software development is a very complex process, usually resulting in a multitude of errors. The detection and removal of these software errors often consumes over half of all software development resources. In an effort to learn more about the software errors being introduced into their products, many organizations are attempting to collect and maintain software error data. These software error data collection efforts can provide needed information about the software development process.

One major application of these software error data is evaluating various tools, techniques, and methodologies developed by software engineers over the last 15 years. Since a major purpose of many of these tools and techniques is to increase product reliability and reduce its cost by decreasing the number and severity of errors, software error data collection provides a natural means for validation of these claims.

In addition to evaluating software engineering tools and techniques, error data may assist in the evaluation of reliability and productivity metrics. These data may also be useful in guiding project managers in resource allocation if certain types of errors are shown to occur more frequently in certain types of software units, or if certain tools are shown to reduce the number or severity of errors likely to occur in a project. Finally, error data may help guide the development of new tools by indicating weak areas.

In this paper the results of an extensive literature review of current software data collection schemes will be summarized. The weaknesses and inabilities of these schemes to satisfy all the potential benefits of data collection will be noted. A new causative software error classification scheme will then be proposed to overcome these deficiencies.

REVIEW OF ERROR CLASSIFICATION SCHEMES

In order to develop a useful error classification scheme, a comprehensive literature review of existing error classification schemes was performed. Although details of this study are contained in another publication, a summary is given here. Errors have been classified by symptom, by cause, by the type of code in which a fault was found or a change was made, by the project phase in which the error was introduced, by severity, or (usually) by some combination of these.

A symptomatic error classification attempts to form groups describing the faults in a document or piece of code. These categories may be broad (e.g., logic, data handling, interface), or specific (e.g., incorrect testing of a loop condition, uninitialized variable). The data for this type of scheme are relatively easy to collect from a typical problem report, that is, a document describing some problem experienced with the software. Such a scheme is most useful in measuring the relative effectiveness of different error detection techniques. The major problem with a system as detailed as Endres is that many of the categories are project-specific and would disappear with a different type of project, set of hardware, and source language.

Causative error categories attempt to identify what should have been done differently in order to prevent the error. This sort of classification is more useful in evaluating software engineering tools, techniques, and measures. Endres has shown how a causative scheme may be derived from a symptomatic scheme. The same sort of technique was used in the TRW Software Reliability Study. It must be noted, however, that this mapping is neither one-to-one nor onto. The common source of error data when a causative classification is used is the change report, a configuration control document, completed by the person who figured out what caused a particular fault and how to fix it. Change reports typically indicate the change, but not the source of the problem. One problem with using the change report to make inferences about the source of an error, however, is that the change selected may have been chosen, not because it fixed the error, but because it was the easiest way around the problem. This is likely to occur, for example, when requirements ambiguities or high-level design omissions are not discovered until late in development.

Errors are often classified by the type of code in which a fault or change occurs. This can provide useful feedback to programmers and testers for the purpose of concentrating their efforts. Classification of errors by the project phase in which the error was committed is natural because the different activities associated with each phase are bound to introduce different types of errors. Thus, almost every error data collection scheme takes into account project phase.

Another type of classification scheme is based on the severity of the error, severity being defined by time, cost to fix, or impact on the system. Error severity, when measured by the impact the error has on the system, is usually divided into three categories—serious, major, and minor—where a serious error caused a system crash or prevented the accomplishment of a primary system objective, a major error noticeably reduced system performance, and a minor error was transparent to the user.

To summarize, errors have been classified by symptom, by cause, by location of fault or change, by source development phase, and by severity. Most of the error data collection schemes involve a combination of two or more of these five types of schemes. All these categorizations can provide useful
information for programmers, testers, managers, and researchers if the data are reliable. For the data to be reliable, errors must be classified by the person who has analyzed the problem and fixed the error. Because of this, the classification scheme must be easy to understand and use. The importance of primary importance to researchers is that of error cause, which is also the most difficult to collect. For conclusions to be drawn about software in general, data must be collected from a variety of real-world software development environments.

PROPOSED ERROR CLASSIFICATION SCHEME

Analysis of previous software error data collection efforts and discussion with software quality assurance personnel in several companies yield some principles for the design of a new error data collection effort. (1) Data collection in the real world must be minimally obtrusive into a company's normal procedures in order to minimize expense and the effects of data collection on the data, and to maximize the reliability of the data. There are two principal effects of the data collection process on data: the time spent collecting the data detracting from the programming effort, and the Hawthorne effect—that is, the knowledge that a particular aspect of a task is being monitored affecting how it is performed. (2) It is desirable to collect as many pieces of information as possible to allow analysis of controlling factors, find unexpected results, and provide for future use of the data in as yet unformulated experiments. It is good scientific practice to record anything that could conceivably have an effect on the results of an experiment. This is especially true when studying complex human behavior and products. The expense of software development and maintenance mandates data collection, and the expense of the data collection mandates efficient use of the data. (3) The data must cover as many different environments and projects as possible to validate any claims about software in general. This is true as long as we are unaware of many controlling factors and cannot control many of the ones we are aware of. (4) The effort must maximize both long- and short-term benefits for the data collectors, their managers, their companies. Data should be collected into a database that will help them concentrate their efforts on problem-prone areas of the current project and know the likely effect of various choices on software error content. (5) The new data collection effort should complement previous efforts to verify or refute their results for different environments, to allow use of their data in conjunction with the new in other experiments, and to obtain types of data that their effort failed to collect. In the remainder of this section a Software Change Report form designed with these principles in mind will be described.

A cursory examination of the Software Change Report form in the appendix reveals some important features of its design. The report differs from typical change reports in that it is designed specifically to deal with changes due to discovered mistakes. Changes due to newly requested capabilities, the lack of which cannot be attributed to the software development organization, should not be considered in software reliability studies or other error analyses. As with other change reports, it is intended to be completed by the person who diagnosed the problem and prescribed the cure, since this person will have the most insight into the type of information required. To make data collection as painless as possible for this often harried person, all responses are simple checkmarks and codable short answers. The elimination of prose responses will make data collection quicker and more reliable and analysis less expensive. It also permits data collection to be totally automated, with interactive error reports connected directly to the database.

This form is derived from the change report form developed by Ostrand and Weyuker, who derived theirs from Weiss. It differs mainly in two respects: (1) wording was neutralized so as to reflect as nearly as possible a consensus of current software development practices, rather than favoring any one company; (2) causative and symptomatic error classifications replace the prose responses, thus minimizing the time required to collect the data without biasing it with a researcher's interpretation for classification.

The approach taken here differs from that of Basil and Weiss in that the report form was designed to handle all phases of the software life cycle. Providing a single form for use during all stages of development and maintenance will simplify the data collection process and reduce the time required to train personnel in its use. Though it is expected that the report form will evolve somewhat with changing research needs and refinements of the classification schemes, the constant use of a single, general-purpose form will essentially eliminate the Hawthorne effect after an initial startup period. Nevertheless, on the advice of Weiss, each item on the change report was selected after brainstorming for research questions and error-related data needed to deal with them.

The first part of the Software Change Report contains configuration management data, error severity data, problem detection and isolation methods used, problem detection and isolation time estimates, a symptomatic error classification, and change properties of the error. This information is not new and is described in more detail in a technical report. In the remainder of this section, emphasis will be placed on the causative nature of the error classification scheme.

The causative error classification is a two-dimensional scheme accounting for both the software life cycle phase during which the error originated and the behavior or conditions that caused the error. One problem with most software error classification schemes is that they fail to recognize that different kinds of errors are made in different phases of the software life cycle. This scheme was derived by noting the kinds of activities occurring and documentation produced in each phase of the life cycle, and then analyzing the kinds of errors that could be made, given the activities.

Recognizing that different companies divide the software life cycle differently, and that they organize the information produced at each stage into different types of documents, the first step in developing the two-dimensional causative error classification scheme was to define life cycle phases by the kinds of information produced. The intention here is not to dictate to a company how it should organize its software development and maintenance processes. Rather, the intention is to allow each software organization to identify which activities that they are already performing belong in the phases of the
classification scheme. This way, meaningful comparisons of data between companies can take place without disrupting their operation.

For the purposes of this error data collection scheme, then, the software life cycle phases are defined by a set of activities and the resulting documentation. The Requirements Specification stage involves the communication and definition of functional and performance requirements. The documents produced in this stage may be thought of as a contract specifying what a system is supposed to do from the point of view of the designers, the testers, the customer, and the users.

The High Level Design phase involves the analysis of requirements and definition of a solution that will meet these requirements. The resulting documentation specifies the components of the system that will perform each required function and define how the various components are supposed to interact.

Detailed Design and Coding involve the implementation of component specifications with machine-readable algorithms and data specifications. The resulting documents include, at the very least, compiled source listings, and often, explanatory prologue and higher-level algorithms in pseudocode, PDL, or flow chart form. Although low-level design is often considered a separate task from coding, both activities are characterized by the successive refinement of algorithms, and the types of errors possible are essentially the same. Errors made at this stage are mostly logical and arithmetic in nature.

One stage of software development appears to be missing from this scheme: testing and verification. The reason for this is that this error classification scheme is based on types of documentation produced, rather than a distinct period of time. Errors in the software that originate while a programmer is doing informal unit testing should be included in Low Level Design and Coding, and errors that occur after a component has been submitted for integration should be included under Debugging and Maintenance.

The final category in this dimension refers to additional errors which, because of the availability of preexisting documentation covering all development phases, may occur during Debugging and Maintenance. While all activities falling after the initial release of a software product have traditionally been lumped into a single category called Maintenance, these activities are usually similar to those of initial development. As in initial development, the requirements for a change and the environment must be understood, a design must be developed and implemented, the system must be retested, and documentation must be written to explain the change within the context of the system. The main difference is that there is some preexisting documentation, which may be used and must be maintained if it is to be used in the future.

The second dimension of this scheme consists of causative error categories for each phase. These are sorted into three rough groups corresponding to steps in the problem-solving process: (1) Understanding the task in all its dimensions: Communicational difficulties will cause errors of misunderstanding. (2) Analyzing the problem and synthesizing a solution: Conceptual difficulties will cause errors at this stage. (3) Documenting the solution: If the solution is going to be used by other people, it must be documented in such a way that the whole solution is clear. Clerical errors are the result of oversights in the process of documentation. Often, tools are devised to assist in one or another of these areas; therefore, while the categorization of errors at that level is rough, a concentration or dearth of errors in one of these rows may indicate the need for better tools to eliminate those kinds of errors or the success of a particular tool.

The error categories are described in detail with examples in another report. A brief explanation of each error category is presented in Figure 1.

### Requirements Specifications

<table>
<thead>
<tr>
<th>Specification of requirements for function and performance in a new system or in a change to an existing system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
</tr>
<tr>
<td>Inadequate understanding of programming environment</td>
</tr>
<tr>
<td>The programming environment includes personnel, machinery, management, development tools, and any other factors which affect the programmer's ability to implement the requirements.</td>
</tr>
<tr>
<td>Inadequate understanding of implementation environment</td>
</tr>
<tr>
<td>The implementation environment includes the configuration of the system the new software is to replace, if any, and any interfacing machinery, software, personnel, or procedures.</td>
</tr>
<tr>
<td>Inadequate understanding of customer's requirements</td>
</tr>
<tr>
<td>Totally missing requirements are caused by an incomplete understanding of the customer's needs. The customer may not think to mention every detail because he figures that the professional knows what details are relevant, and that he will ask appropriate questions.</td>
</tr>
</tbody>
</table>

### Miscommunication of capabilities to customer

Failure to explain the capabilities of a computing system may lead to an unsatisfactory product, with the subsequent need for enhancements.

- Inadequate communication between team members
- Confusion between team members regarding responsibilities or decisions

### Conceptual

Unnecessary requirement

- Specification of a requirement the customer did not ask for.

### Untestable requirement

Specification of a requirement whose fulfillment cannot be verified.

---

Figure 1—Error categories
<table>
<thead>
<tr>
<th>Inconsistent requirements</th>
<th>Insufficient capacity of design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification of conflicting requirements.</td>
<td>System architecture which cannot meet a performance requirement when implemented.</td>
</tr>
<tr>
<td>Unimplementable requirement</td>
<td>Clerical</td>
</tr>
<tr>
<td>Specification of a requirement which cannot be fulfilled, given the state of the art and the system configuration.</td>
<td>Vague specification</td>
</tr>
<tr>
<td>Undocumented requirement</td>
<td>Not enough detail employed in the specification of a component.</td>
</tr>
<tr>
<td>Omission of a requirement from one of the documents.</td>
<td>Nonmatching interface specifications</td>
</tr>
<tr>
<td>Omission of a requirement from one of the documents.</td>
<td>Number or type of passed elements not matching.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clerical</th>
<th>Nonadherence to standards</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clerical</td>
<td>&quot;Standards&quot; refers to standards of documentation enforced in a particular software house.</td>
</tr>
<tr>
<td>Other</td>
<td>Undocumented assumption</td>
</tr>
<tr>
<td>Typographical and other miscellaneous errors.</td>
<td>This category is for errors caused by designers making conflicting assumptions about interacting features because they were not documented.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High Level Design</th>
<th>Detailed Design and Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification of system components to fulfill requirements. System design.</td>
<td>Refinement of the design for each component and actual implementation.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Communicational</th>
<th>Conceptual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inadequate understanding of programming environment</td>
<td>Unspecified feature added</td>
</tr>
<tr>
<td>Same as above.</td>
<td>The addition of a feature not required by the high level design.</td>
</tr>
<tr>
<td>Inadequate understanding of implementation environment</td>
<td>Specified feature missing</td>
</tr>
<tr>
<td>Same as above.</td>
<td>The omission of a feature clearly specified in the high level design.</td>
</tr>
<tr>
<td>Misunderstanding of requirements</td>
<td>Unnecessary logic</td>
</tr>
<tr>
<td>Inability of designer to understand a (set of) requirement(s), not traceable to a specific flaw in a Requirements Specifications document.</td>
<td>Extra logic which reduces performance and does not improve comprehensibility of source code.</td>
</tr>
<tr>
<td>Inadequate communication between team members</td>
<td>Missing, incorrect, or inconsistent logic</td>
</tr>
<tr>
<td>Same as above.</td>
<td>A logic sequence which does not correctly implement a feature.</td>
</tr>
</tbody>
</table>

**Figure 1 (continued)**
The following subcategories of incorrect logic should be used when applicable.

**Timing error in concurrent modules**
- Flow control which results in deadlock, problems in accessing a critical section, or problems in the use of information deposited in a critical section.

**Imprecise numerical approximation**
- Erroneous numerical analysis technique.

**Missing exception handling (robustness)**
- Failure to provide for an exception condition.
- Insufficient capacity of machine implementation
  - Module implementation which does not meet a performance requirement.

**Clerical**
- Data definition not conforming to specification
  - Inaccurate transcription of data specification into implementation language.
- Interface not conforming to specification
  - Inaccurate transcription of interface specification into implementation language.
- Uneducated use of programming language features
  - Erroneous or poor implementation due to the programmer not taking advantage of a programming language feature.

**Nonadherence to standards**
- Same as above.

**Undocumented reasoning for difficult or unusual solution**
- Neglected documentation may cause problems when referencing or changing the source code.

**Undocumented assumption**
- Same as above.

**Other**
- Same as above.

**Additional Errors incurred by altering existing products**
- During debugging and maintenance, these errors may occur because of the possible existence of all levels of documentation.

**Communicational**
- Existing documentation inconsistent
  - Error caused by disagreement between two or more pieces of documentation.
- Existing documentation incomplete
  - Error caused by a lack of information about requirements or earlier design decisions.
- Misunderstanding of documentation
  - Inability of programmer to understand existing documentation, not traceable to a specific flaw therein.
- Misinterpretation of language constructs
  - Failure to understand source code due to lack of familiarity with the programming language.

**Conceptual**
- No categories.

**Clerical**
- Revisions not made to all documentation
  - Failure to update all documentation.

---

**CONCLUSION AND FUTURE RESEARCH**

In this paper a causative software error classification scheme has been presented. This scheme differs from previous software error classification schemes by emphasizing the entire software life cycle and the causative attributes of the errors.

Data carefully collected according to this scheme can be used to validate the claims of many software engineering tools and techniques.

Some limited experimentation with this error scheme has shown that classification of an error by life cycle phase and major causative categories is not difficult. More detailed classification within a single major category does, however, require more training. We plan to continue refining and experimenting with the causative error classification scheme. We also plan to attempt to calculate some measure of reliability for the error classification scheme that reflects its ability to capture accurate data. We will then attempt to use this measure to compare the reliability of existing schemes. This comparison must also take into account the differences in resolution of the different techniques.
REFERENCES


APPENDIX

<table>
<thead>
<tr>
<th>SOFTWARE CHANGE REPORT</th>
<th>No. ______________</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project:</td>
<td>Date problem found: ______________</td>
</tr>
<tr>
<td>Data problem isolated:</td>
<td>Date problem fixed: ______________</td>
</tr>
<tr>
<td>Unit(s) requiring change:</td>
<td></td>
</tr>
<tr>
<td>Check all appropriate boxes:</td>
<td></td>
</tr>
<tr>
<td>RETEST:</td>
<td></td>
</tr>
<tr>
<td>Data:</td>
<td>[ ] Passed [ ] Failed, same symptoms</td>
</tr>
<tr>
<td>[ ] Failed, new symptoms</td>
<td>New report id: ______________</td>
</tr>
<tr>
<td>SEVERITY:</td>
<td></td>
</tr>
<tr>
<td>[ ] Prevents accomplishment of primary function, jeopardizes safety</td>
<td></td>
</tr>
<tr>
<td>[ ] Degrades performance or maintainability, with no workaround</td>
<td></td>
</tr>
<tr>
<td>[ ] Degrades performance or maintainability, but a workaround exists</td>
<td></td>
</tr>
<tr>
<td>[ ] No apparent effect on performance or maintainability</td>
<td></td>
</tr>
<tr>
<td>Detection Mechanism</td>
<td>PROBLEM DETECTION</td>
</tr>
<tr>
<td>Group Requirements Review</td>
<td>Methods used</td>
</tr>
<tr>
<td>Group Design Review</td>
<td>[ ]</td>
</tr>
<tr>
<td>Programmer Desk Checking</td>
<td>[ ]</td>
</tr>
<tr>
<td>Other Person Desk Checking</td>
<td>[ ]</td>
</tr>
<tr>
<td>Group Code Review</td>
<td>[ ]</td>
</tr>
<tr>
<td>Test Runs by Programmer</td>
<td>[ ]</td>
</tr>
<tr>
<td>Test Runs by Other Person</td>
<td>[ ]</td>
</tr>
<tr>
<td>Program Failure for User</td>
<td>[ ]</td>
</tr>
<tr>
<td>Other:</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

Time used to: < 1 hour 1 - 4 hours 5 - 8 hours 1 - 5 days > 1 week: est.

| isolate problem | make correction | |
| [ ] | [ ] | [ ] | [ ] | [ ] |

FAULT in: [ ] Data definition [ ] Data handling [ ] Test [ ] Documentation [ ] Test plus processing [ ] Environment [ ] No fault incurred

Type of data involved: [ ] Address [ ] Control [ ] Primary data

Type of test involved: [ ] Loop control [ ] Branch control

Something was: [ ] Omitted [ ] Superfluous [ ] Incorrect

CHANGES were made to: [ ] no documents [ ] 1 - 5 instructions in 1 routine [ ] > 5 instructions in 1 routine [ ] > 1 routine [ ] > 1 routine in 1 module [ ] > 1 module documentation in code [ ] user manual [ ] low level design [ ] data base [ ] Interface specification [ ] high level design [ ] requirements specification

| Other: [ ] | |

The changed document(s) was/were: [ ] written entirely by you [ ] written partly by you [ ] not written by you at all

From the collection of the Computer History Museum (www.computerhistory.org)
### A Proposed Causative Software Error Classification Scheme

#### Steps of development
- Requirements specification
- Feasibility study
- High level design
- Low level design
- Coding
- Unit test
- Integration test
- Program acceptance test
- System test/Beta-site test
- After product release
- Unknown

#### Fault first noticed

| Requirement Specification | High Level Design | Detailed Design and Implementation | Additional Errors
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inadequate understanding of programming environment</td>
<td>Inadequate understanding of programming environment</td>
<td>Misunderstanding of module or data definition</td>
<td>Existing documentation incomplete</td>
</tr>
<tr>
<td>Inadequate understanding of implementation environment</td>
<td>Inadequate understanding of implementation environment</td>
<td>Misunderstanding of requirements</td>
<td>Misunderstanding of documentation</td>
</tr>
<tr>
<td>Inadequate understanding of customer's requirements</td>
<td></td>
<td></td>
<td>Misinterpretation of language constructs</td>
</tr>
<tr>
<td>Miscommunication of capabilities to customer</td>
<td>Inadequate communication between team members</td>
<td>Programmer unaware of convention</td>
<td></td>
</tr>
<tr>
<td>Miscommunication of capabilities to customer</td>
<td>Inadequate communication between team members</td>
<td>Misinterpretation of language constructs</td>
<td></td>
</tr>
</tbody>
</table>

#### Error Cause and Source Phase: Locate the column, row, and individual category.

- Communication
  - Unnecessary requirement
  - Inadequate implementation
  - Inconsistent requirements
  - Unimplemented requirement
  - Undocumented requirement

- Conceptual
  - Missing feature
  - Unnecessary feature
  - Unspecified feature
  - Inconsistent design
  - Incorrect or inconsistent logic

- Clerical
  - Nonconformity to standards
  - Vague specification
  - Nonmatching interface specifications
  - Undocumented assumptions
  - Nonconformity to standards

- Unknown

#### Other

- [ ] Error occurred
- [ ] Fault first noticed
- [ ] Requirement specification
- [ ] High level design
- [ ] Low level design
- [ ] Coding
- [ ] Unit test
- [ ] Integration test
- [ ] Program acceptance test
- [ ] System test/Beta-site test
- [ ] After product release
- [ ] Unknown

#### Additional Errors

- [ ] Existing documentation incomplete
- [ ] Misunderstanding of documentation
- [ ] Misinterpretation of language constructs
- [ ] Misunderstanding of documentation

#### Notes

- Errorenous fix of a previous error
- Old report id
- Unknown cause

---

From the collection of the Computer History Museum (www.computerhistory.org)