Chapter 5. Identifying Mistakes Causing Defects

5.1 Causal analysis 2
5.1.1 Types of mistakes 2
5.1.2 Forms of causal analysis 2
5.1.2.1 Author questioning 3
5.1.2.2 Error supposition 3
5.2 Discovering software-related defects 3
5.2.1 Sibling search 4
5.2.2 Metamorphosis detection 5
5.3 Identifying mistakes causing failures 5
5.4 Psychology of causal analysis 8
5.5 Pragmatics of causal analysis 9
Bottom line 9
Supplementary readings (References) 9

Testers can do more than discover symptoms of human error i.e., implementation defects, they can discover the causes of these defects

While we emphasize discovery of requirements defects and their causes, we now broaden our focus to consider discovery of design defects and causes and causes of implementation defects. Broadening is fitting because the analysis goals and tactics are the same and because requirements defects may propagate to cause design or implementation defects.

The terms **mistake** or **error** are used to refer to a human action that is misguided or wrong. **Defect** refers to a fault or flaw in a software-related work-product e.g., plans, requirements, design, code, data, or procedures. **Failure** refers to lack of success.

We now consider the causal analysis following defect discovery and the challenges of failure analysis.

5.1 Causal analysis

We want to understand the reasons for mistakes that cause software-related work-product defects. This understanding enables the discovery of other defects caused by the same type of mistake. For example, inadequate understanding of quality attribute requirements causing a defective security requirement may also cause a defective safety requirement. In addition, understanding these mistakes can help to identify mitigations for some of them.

5.1.1 Types of mistakes

We now describe different types of mistakes that can be made during the development of any software-related work-product.

Mistakes can be grouped as planning or execution. **Planning mistakes** can be caused by *inadequate understanding* of domain, tasks, contexts, information sources, major risks, requirements, or solutions. They can also be caused by product constraints e.g., cost, project practices or constraints e.g., inadequate time, deliberate violations of standards or regulations, or by temporary loss of memory or focus.

Inadequate understanding is caused by inadequate training or experience (e.g., inadequate understanding of interface design), inadequate or incorrect information (e.g., incorrect assumptions or precursors), or misinterpreting or ignoring correct information.

Execution mistakes are forced or unforced. Forced mistakes are caused by inadequate time or task overload. Unforced mistakes are caused by temporary loss of memory or focus (e.g., caused by lack of sleep) or by inadequate sharing of information or ineffective participation e.g., in reviews.

5.1.2 Forms of causal analysis

There are at least two forms of causal analysis, direct and indirect. The direct form entails asking a defect author or contributor, why a mistake was made. The indirect form entails using the characteristics of the defect to guess why a mistake was made.

5.1.2.1 Author questioning

A list of questions tailored to the work-product and author or contributor's role should be used to support all forms of questioning. Section 4.2 suggests questions to be used for authors and contributors of requirement's defects. See the sample questionnaire in Appendix A.

Questions can be asked face-to-face, via interactive communication such as telephone or email with an editable file attachment, or by automated survey. Note that email and automated surveys can be followed up by additional questioning.

Author questioning can be supported by any automated survey application containing questionnaires for each of the authors or contributors associated with requirements, design, or implementation defects.

A better solution would be to use a defect or issue tracking system that provides and manages these questionnaires. Such a tracker could notify bug authors and contributors of the need for their help in identifying the causes of their mistakes.

5.1.2.2 Error supposition

Author or contributor questioning may not be feasible because they are unknown, unavailable, or uncooperative. When questioning is infeasible, error supposition can be used to guess the mistakes that caused the defect. Guesses can be based on the type of work-product, the stakeholder's role, the type of mistake e.g., omission, and the subject of the mistake e.g., exception handlers. Error taxonomies, such as the one for requirements (see section 4.2), can aid supposition. The success of searches for other defects based on supposed mistakes would partially confirm error suppositions.

5.2 Discovering software-related defects

Defects in requirements, designs, and implementations can be discovered in many ways such as technical reviews, static analysis, testing, formal methods, and runtime monitoring. Chapter 6 describes the use of skeptical technical reviews and static analysis to detect

requirements defects. Research[1] [Ref. 5.1] suggests that an error taxonomy specific to a software work-product can improve the effectiveness of a work-product review.

We now describe two other activities that lead to defect discovery, sibling search and metamorphosis detection.

5.2.1 Sibling search

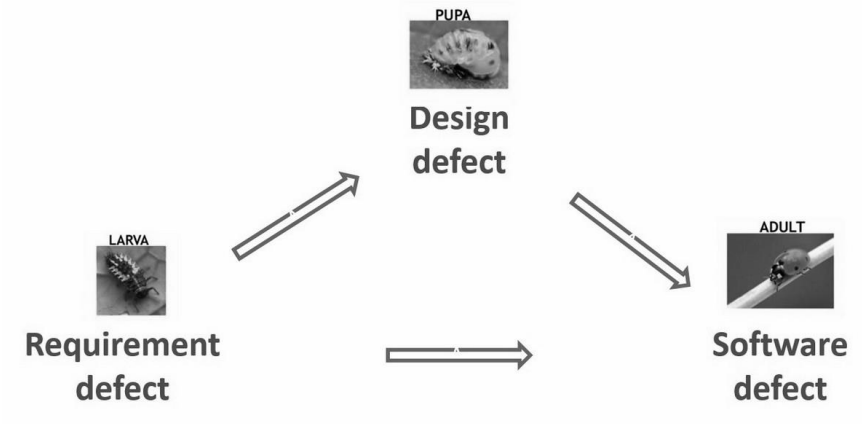
Sibling search is a defect discovery tactic that begins when a defect is discovered by any method. The goal is to find other defects that are *similar* i.e., have similar characteristics, to the triggering defect already discovered.

There are two types of sibling search, identical twin searches and fraternal twin searches. Identical twin searches look for defects with nearly identical characteristics in other locations. These searches can be done manually or automatically [Ref. 5.3].

Fraternal twin (FT) searches look for other defects likely to be caused by the same misunderstanding that caused the triggering defect. For example, finding out that a developer has an inadequate understanding of a programming language construct could trigger an FT search for other constructs that are difficult to understand. Authors who focus on the reasons for their mistakes are likely to be most effective at FT searching.

[1] Research [Ref. 5.2] also suggests that knowledge of an error taxonomy for requirements helps requirement authors avoid some mistakes i.e., is an effective general mitigation tactic.

5.2.2 Metamorphosis detection

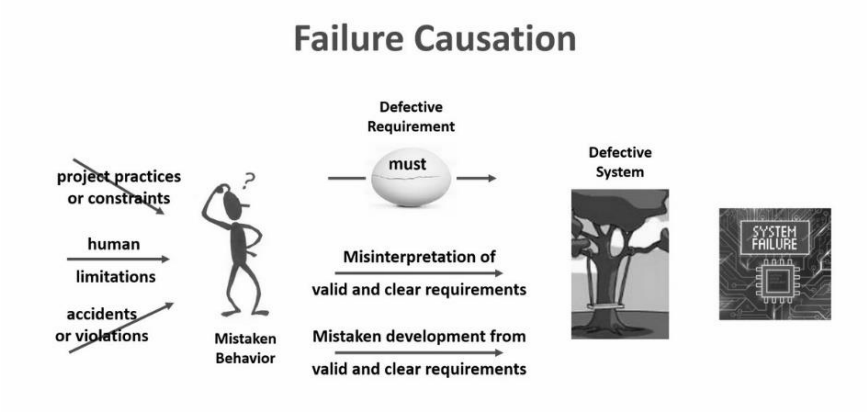


A software defect may have been caused by a lack of sleep or by a design or requirements defect. The challenge is to determine if a downstream defect was caused by an upstream defect. The determination can be made by asking the downstream author or by locating the corresponding requirement or design element and checking them for defects.

5.3 Identifying mistakes causing failures

"These results suggest that the sources of safety-related software *defects* lie farther back in the software development process (in inadequate requirements) whereas the sources of non-safety-related *defects* more commonly involve inadequacies in the design phase."

– Robyn Lutz [Ref. 5.4]



Working backwards from a failure can be very hard

When defective work-products cause a major system failure, the following **failure analysis** process identifies the mistakes causing the defective work-products. A **major system failure** is one that should never happen in production.

A. When developers agree that a major failure has occurred in test or production, investigate as follows:

1. Testers describe the software failure and its effects (e.g., in a bug/issue tracker).
2. Developers identify implementation defects and correct them.
3. Testers use sibling search to locate other implementation defects.
4. Developers correct any added defects.
5. Testers validate all corrections.
6. For each defect, developers rate the scope of repair as **crosscutting** (i.e., impacting many modules) or **local** in a causal analysis database
7. Developers or testers determine if requirement or design elements caused the implementation defect using metamorphosis detection (see 5.2.2) and if so enter element identifiers into a causal analysis database.
8. Developers *identify mistakes causing the corrected defects* (e.g., using the error scheme in 5.1.1) and enter findings into a causal analysis database.

9. If design defects are involved:
    a. defects are corrected
    b. sibling search is used to locate other design defects
    c. additional defects are corrected
    d. all corrections are validated
    e. Designers *identify mistakes causing the corrected defects* (e.g., using the error scheme in 5.1.1) and enter findings into a causal analysis database
10. If requirements defects are involved the:
    a. defects are corrected
    b. sibling search is used to locate other requirements defects
    c. additional defects are corrected
    d. all corrections are validated
    e. defective requirements are typed (e.g., using the table in 2.2.1)
    f. defects are typed (e.g., using the defect scheme in 4.1)
    g. Specifiers *identify mistakes causing the corrected defects* (e.g. using the error scheme in 4.2) and enter findings into a causal analysis database

B. Analyze findings from multiple failures for causal patterns and potential mitigations.

C. Implement and monitor results of approved mitigations.

A **causal analysis database** contains developer error surveys, designer error surveys, and specifier error surveys tied to a single failure. It is understood to be a quality improvement tool, not a management tool. Its contents are private.

The goal of failure analysis is to identify causal defects and their siblings, correct and validate the defects, determine the mistakes causing the defects, and mitigate these mistakes when possible. Steps 8, 9, and 10 will be the most challenging since most of the defect causes will entail human errors of various kinds.

Requirement and design work-products can be in three states: sound, defective, or ambiguous. We use ambiguity to imply that some interpretations are sound, while others are defective. Ambiguity makes

causal analysis more difficult. Was a problem caused by the author of an ambiguous work-product, its user, or both?

The effectiveness of the failure analysis process described above depends on the following **assumptions**:

- Participants, including leaders, are competent
- Work environment is a psychologically-safe, learning organization
- Participants are committed to exploring ways to improve software quality
- Skeptically reviewing upstream work-products is encouraged

5.4 Psychology of causal analysis

The challenge of accurate causal analysis with author questioning is psychological, not technical. Success in getting people to openly and accurately describe their mistakes depends on the attitude of the authors and contributors, the psychological safety of the project environment, and the trust relationship between authors and testers. Success also depends on the interpersonal skills of the tester in delivering the bad news and eliciting causal information. Some people are open and even anxious to learn from their mistakes, others not so much. In all situations, specific information about individual causation must be kept **confidential**.

Establishing a trusting relationship is crucial to success. A work-product author or contributor may feel the real cause of an error is too embarrassingly silly and therefore claim not to remember or make up something. Only a trusting relationship with non-judgmental listening has a chance of learning the truth. Success also requires close cooperation between project stakeholders. All parties must understand the goal is to identify mistakes so they can be avoided or easily detected. The goal is not to blame or punish.

In toxic work environments or with defensive contributors, causal analysis may only work with error supposition. Even in non-toxic work environments with willing stakeholders, causal analysis may not always succeed.

5.5 Pragmatics of causal analysis

Unfortunately, there will be many opportunities for requirements-focused causal analysis. Therefore, if you are in doubt about the cause of a defective requirement and its suitable mitigation, wait for confirmation. A similar problem is likely to occur.

In theory, error supposition (5.1.2.2), metamorphosis detection (5.2.2), and causal analysis (5.3) is helped by comprehensive requirements tracing. Trace information can be used to backtrack from a software defect to its associated design and requirements information.

Unfortunately, manual requirements tracing is tedious and unreliable. In addition, tracing has poor tool support and degrades over time without maintenance to reflect software evolution. Thus tracing is rarely practiced unless required by conformance to a standard or regulation.

Causal analysis can help organizations:

- profile their requirements defects, causal mistakes, and suitable mitigations
- focus on mitigating defects causing major production failures and expensive rework
- inspire requirements process changes

Note that successful mitigation of requirements-based defects will make test failures less frequent and therefore make software defects more difficult to detect by testing.

Bottom line

Without causal analysis, preemptive testing will be unguided or misguided. Accurate causal analysis is a tricky business. People may not admit their mistakes or be willing to analyze them. This is the biggest challenge in preemptive testing.

Supplementary readings (References)

Papers are available at

5.1 Anu, V., Walia, G., Hu, W., Carver, J., Bradshaw, G. (2016) "Effectiveness of Human Error Taxonomy during Requirements Inspection: An Empirical Investigation"

5.2 Hu, W., Carver, J., Anu, V., Walia, G., and Bradshaw, G. (2017) "Defect Prevention in Requirements using Human Error Information: An Empirical Study"

5.3 Guangtai Liang, Qianxiang Wang, Tao Xie, Hong Mei, (2013) “Inferring Project-Specific Bug Patterns for Detecting Sibling Bugs”

5.4 Lutz, Robyn R. (1993) “Analyzing Software Requirements Errors (Defects) in Safety-Critical, Embedded Systems”