

Root Cause Analysis as a Guide to SRE Methods

Timm Grams

University of Applied Sciences Fulda
<mailto:Timm.Grams@et.fh-fulda.de>
<http://www.fh-fulda.de/~grams/index.htm>

Abstract: Which Software Reliability Engineering (SRE) methods should be applied during the various phases of the lifecycle of a product? The answer given here centres on learning from errors. The classification and evaluation of methods is strictly based on causal analyses of disasters, accidents and incidents with undesired outcome. The lifecycle model of IEC Standard 61508 has been adopted as a classification scheme. A couple of examples are given. The SRE methods considered here are those of IEC 61508. These methods are dealing with software reliability as well as with the production of highly reliable software. As an example of some more recent proposals Extreme Programming (XP) has been included.

Introduction

The rules and the timely processing order of Natural Software Engineering (NSE) I discovered by watching my students during a programming course in November 2003:

1. In the very moment you have a faint idea of what you are supposed to do start coding. This is called *realisation*.
2. Derive an excerpt from your program. This results in your *concept* and *design*.
3. Then write down the *specification* and define all surprising program properties (heartlessly called bugs) to be features.
4. Convince your customer (the instructor's role) of what you are able to deliver is what he truly wanted. This challenging task is called *requirements engineering*.

Indeed, this is the natural way: The scheme rests on firm psychological and sociological grounds. For, what you are paid for is real work. Coding is to come first. Whereas activities like haggling over requirements, verification and documentation are introducing delays. And they are real pains. In favour of keeping the time schedule and comfort these activities should be skipped or given low priorities.

On the other hand, our *experience points into another direction*: Disasters happen because of a lack of documentation, badly thought-out specifications, superficial tests and left off verifications. Investigating and analysing the disasters of the past makes the strongest case in favour of the not so natural and painful software engineering methods.

This paper deals with software reliability engineering (SRE) methods. They are more or less painful, and they are more or less useful. The question to be answered is: When does it pay to use them?

The one who has the task of building and maintaining software of automation and safety systems faces a plentiful variety of techniques for software construction and evaluation. Let alone the IEC Standard 61508 lists 66 techniques aiming at software safety integrity - not taken into account all the existing variants of the methods [1, part 7, annex B].

In this situation some guidance is needed. And such will be offered here. This guide comprises three components:

1. A *lifecycle model* serves as the basic classification scheme of SRE methods and techniques.
2. Primary causes of incidents or accidents will be identified by *causal analysis*. Preventive measures are identified.
3. The primary causes can be attributed to lifecycle phases, and this will entail a *classification of SRE methods* suitable for prevention.

The resulting classification (or taxonomy) of SRE methods should help starting up SRE activities and to bring into focus the most effective methods and techniques for solving the actual problems (fig. 1).

The framework of the following considerations is given by the IEC Standard 61508 applicable to electrical, electronic and programmable electronic (E/E/PES) safety-related systems [1].

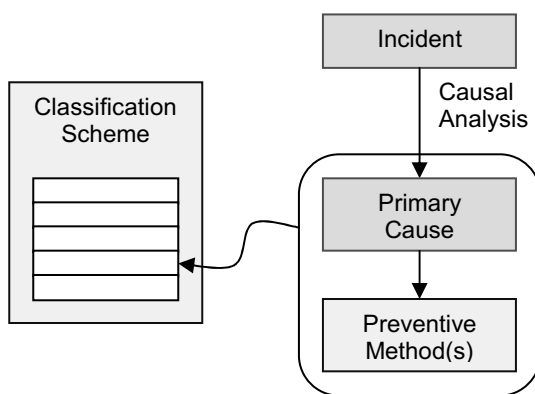


Fig. 1 The proposed procedure

For these systems a *safety plan* shall be prepared at the outset and this plan shall be updated during the entire safety lifecycle. This safety plan shall specify “procedures which ensure that hazardous incidents or incidents with potential to create hazards are analysed and recommendations made such that the probability of repeat occurrence is minimised” [1, part 1, 6.2.2k]. This imperative says how the *learning from errors* shall be institutionalised, and it also draws the lines along which this paper will evolve.

Root Cause Analysis

A Three-level Model of Causality

There are different views on the causes of accidents and incidents. In this paper three levels of causal analysis will be distinguished, fig. 2:

1. *Analysis of Immediate Causes*. This analysis is based on the „logic of causality“: An (immediate) cause can be conceived to be an element of a set of causes, each of them *necessary* and all of them *sufficient* to cause the undesired event or accident [3, p. 43]. This level of causal analysis concerns the technical blow by blow, the chain of events leading to the accident. The most predominant causes will be called *primary causes*. In the following causes with some relevance to the software of automation and safety systems will mainly be considered.

2. *Detection of Errors.* A cause may be an error, or may not. An error may be a cause, or may not. Causal analysis can be used for identifying possible causes. Beyond that a normative model is needed to distinguish errors from „normal“ causes [2].

3. *Root Cause Analysis.* It is worthwhile to find out the general mechanisms leading to an error. Such general mechanisms have been identified and described by the technical, psychological and sociological sciences. Root cause analysis aims at maximum learning from errors through *generalization* [3, chapter 3 and 4].

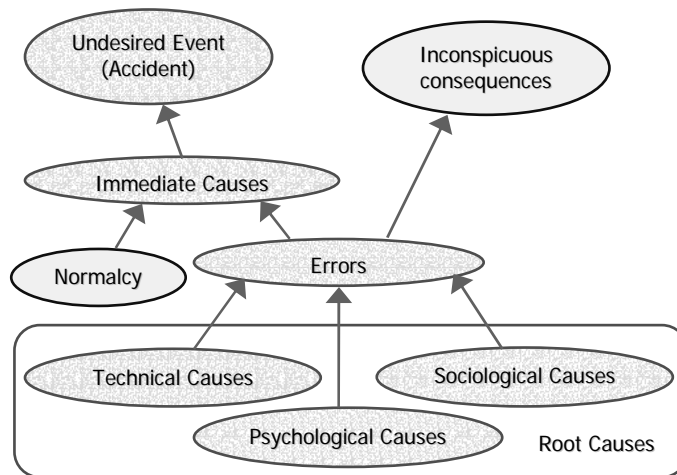


Fig. 2 Causes and Errors

The Lifecycle Classification Scheme

The classification scheme of table 1 has been devised in accordance with the lifecycle model given in IEC Standard 61508 [1, part 1, chapter 7] and it is similar to the one from the “Out of Control” publication of the Health and Safety Executive (UK) [4].

This classification scheme will be used to classify the root causes of incidents/accidents and preventive methods as well. The classification procedure will be carried out for a number of cases with undesired outcome. The case descriptions are based on actual accidents or incidents. In this paper all these cases are called *incidents*.

Every incident under consideration will be described and analysed in compliance with the following scheme.

Table 1 Classification Scheme (Lifecycle)

Phase	Description
1	<i>Specification</i> - Functional Requirements - Safety Integrity Requirements
2	<i>Realisation</i> - Design - Implementation (Coding) - Validation and Verification
3	<i>Installation and Commissioning</i>
4	<i>Operation and Maintenance</i>
5	<i>Changes after Commissioning</i> - Modification, Reuse and Retrofit - Decommissioning

Title: Denomination of the accident or incident.

Source: A bibliography leading to more detailed descriptions of the incident and to further references.

Setting: Short description of the scene (system and environment) where the incident occurred.

Course of Events: Short description of the chain of events leading to the undesired consequences.

Root Cause(s): The result of a causal analysis and record of the main arguments: Identification of a primary cause and demonstration that it is indeed an error. (Because we are aiming at learning from errors only incidents resulting from errors are worthwhile to be considered.) Description of the root cause(s) of the error.

Preventive Method(s): List of methods applicable to the prevention of errors identified by root cause analysis.

Classification: Naming the lifecycle phase to which the primary cause can be assigned.

The HSE publication [4] reports on a study during which 34 accidents and incidents due to control system failures have been investigated. For every incident a primary cause has been identified and attributed to one of the lifecycle phases. The main result of the study is given in fig. 3.

Most causes of undesired events originate from wrong specifications. Therefore we will at first clarify the term "Specification Error".

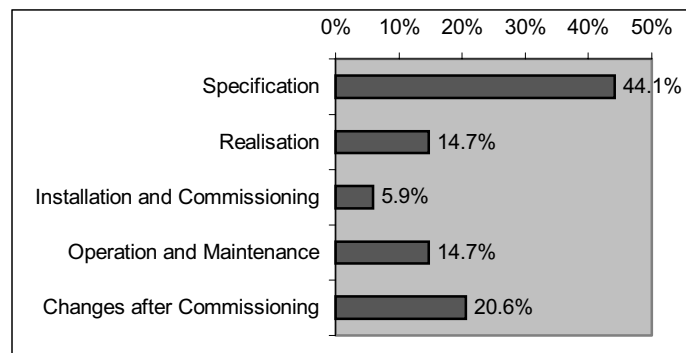


Fig. 3 Results of causal analysis (HSE-study, 34 incidents)

The Meaning of „Specification Error“

Causes are identified *a posteriori* - i.e. after an undesired event has occurred - on the basis of purely formal principles.

Knowing the causes as such is of little help to us. Only a cause classified as an error offers a chance of learning therefrom. To find out whether a given cause (what has happened) can be conceived to be an error we are in need of a normative model (of what should have happened). By definition errors are deviations from a norm or a normative model.

The specification serves as a yardstick and defines the correct behaviour of the system, and from this we do know what an error is [5]. Specifications are some kind of norm with respect to all the following lifecycle phases, the specification phase being excluded.

Specification errors are nonexistent within the scope of a project and its lifecycle phases. The specification complies with itself, and consequently cannot offend against itself.

Thus we are in need of an extension of the notion of “error” if we want to speak of specification errors.

Certainly, there is a goal quite from the outset of a project: To make the risk accompanying the use of the system as low as reasonably practicable. But from this general goal the (safety) functions requirements specification and the (safety) integrity requirements specifications are to be developed and allocated to the system’s components. This decomposition and allocation process cannot be based solely on some “super norms”. *Experience* and *creativity* are indispensable ingredients.

Safety related specification errors come into existence through accidents or incidents with undesired outcome: A root cause analysis may lead to corrections of the specification of the system, constituting the norm of a possible subsequent lifecycle aiming at a safer system.

Specifications are thus subject to evolution. And the new specification taken as a norm is a refutation of the older one. In this sense we are entitled to speak of specification errors. Specification errors are errors *a posteriori*.

The evolution of specifications and systems is analogous to evolutionary processes in biology, science, and society. Let us take a very long neck as part of the currently valid “specification of the giraffe”. Predecessors of the giraffe were assumedly endowed with shorter necks. In hindsight this can be understood to be an error. According to Karl Raimund Popper evolution “should be visualized as *progressing from problems to problems*” [6, p. 222].

Specification errors come into existence by hindsight. And this is why specification errors cannot be taken into account during reliability and safety assessment activities.

Incidents: Analysis and Classification

This section presents two case studies: The Royal Majesty Accident and the Ariane 5 Maiden Flight Failure. They are given as examples to show how the proposed procedure shall be applied. The root causes and the preventive methods derived from these cases are assigned to the lifecycle phases *Realisation* and *Changes after Commissioning*.

Table 2 summarizes the classification of methods. It contains the methods derived from the accidents considered below and additionally those derived from the case studies 1 (*Specification*), 8 (*Installation and Commissioning*), and 9 (*Operation and Maintenance*) given in the HSE publication [4, Section 3].

Further material on control system failures can be found in [11] and [12]. Incidents related to computers are being discussed in the online forum „Computer Risks“, edited by Peter G. Neuman [7].

The Royal Majesty Accident

Source: Marine accident report of the National Transportation Safety Board (NTSB) on the grounding of the *Royal Majesty* in 1995 [8]. The description of the event and causal analysis is based on [9].

Setting: The cruise ship *Royal Majesty* was equipped with an integrated bridge system consisting of an autopilot obtaining position data from a GPS unit.

Course of Events: Shortly after departure from St. George's, Bermuda, on June 9, 1995, the GPS switched to dead-reckoning mode changing the status bit from valid to invalid to indicate that valid position data is no longer being transmitted. The autopilot did not recognise the GPS' dead-reckoning mode and accepted the data as valid. The personnel having the watch relied on the position data of the GPS. The vessel grounded on a shoal. The GPS was checked after the grounding. The GPS was in error by at least 15 nautical miles.

Root Cause(s): The antenna cable of the GPS was routed in such a way that it could be kicked or tripped over. Post-accidental examination of the GPS found the antenna cable of the GPS had separated from the factory connection at the antenna. Subsequently the system has inappropriately dealt with this failure due to a communication problem between the GPS and the autopilot. The autopilot's interface has been designed badly such that not all the failure indications generated by the GPS are recognized. Because our focus is on the automation system this inadequate design is thought a *primary cause* and a *design error*.

What are the deeper causes of this error? The designer of the autopilot expected the GPS to indicate invalid data by (a) nulled position data or (b) halted transmission or (c) no changes in the position. Whereas the GPS used the dead reckoning mode and a valid/invalid bit to indicate invalid position data. Since the designers did not expect the GPS to send position data based on anything other than GPS satellite data they chose not to check the valid/invalid bits in the data stream.

To put it more schematically: Let A be the proposition "GPS sends nulled position data or no changes in the position; or transmission is halted" and let B stand for "GPS indicates invalid data". Let C be the proposition "GPS sends invalid data". Then it can be concluded that "from A follows C ". The designer may have lost the other possibility (from B follows C) out of sight and may have generalized his knowledge (from A follows C) to conclude: from not A follows not C . This *failure to apply modus tollens* in conjunction with the *mind set effect* is a thinking trap well known from cognitive psychology. This thinking trap belongs to the larger class of *inductive reasoning errors* [10].

Preventive Method(s): Communication problems can be solved by the *modular approach*, especially by treating interface descriptions as specifications. For outermost exactness standard mathematical and logical notation should be used. The specification has to be documented for use in the following lifecycle phases, especially the fifth phase (changes after commissioning: modification and reuse). *Failure mode and effect analysis*, FMEA, is an effective means to overcome the mind set effect and the failure to apply modus tollens. By this method the attention will be drawn to all possible deviations from normal behaviour.

Classification: Realisation (design, implementation and verification).

The Ariane 5 Maiden Flight Failure

Source: The course of events as well as the causal analysis is taken from the report by the Inquiry Board on the ARIANE 5 - Flight 501 Failure [14].

Setting: The attitude of the Ariane 5 launcher and its movement in space are measured by an inertial reference system.

Course of Events: On 4. June 1996 the maiden flight of the Ariane 5 launcher ended in a failure about 30 seconds after lift-off. The initiating event has been an unexpected high value of a horizontal velocity sensor which caused overflow of an internal variable of the on-board computer. This variable was one of three variables within the respective software module left unprotected, although comparable variables in the same place in the code were protected from causing an operand error. The design of the Ariane 5 inertial reference system is practically the same as that of the Ariane 4 system. The decision to protect certain variables but not others was that they were either physically limited or that there was a large safety margin. Ariane 5 trajectory data were not included in the requirements and specification of the inertial reference system. The value of the variable was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of the Ariane 4 and results in considerably higher horizontal velocity values.

Root Cause(s): It seemed not wise to make changes in software which worked well on Ariane 4. The engineer's belief that „change is bad“ is thought to be a root cause of the accident.

Preventive Method(s): Formulation and documentation of *specifications* of all modules using standard mathematical notation. Before reuse: *interface analysis* (rigorous check whether all the actual requirements are met by the specification of the module under consideration), *boundary value analysis* and *equivalence classes testing*.

Classification: Changes after Commissioning (Modification and Reuse).

Unconventional methods: Extreme Programming (XP)

IEC 61508 [1] draws on today's software engineering. This may be entailing bureaucracy, tedious procedures and inflexibility with respect to changing requirements. To come up against the drawbacks of standard software engineering a group of computer scientists formulated the "Manifesto of Agile Software Development" [15].

In accordance with these new principles Kent Beck and his colleges and disciples have pushed forward a „new“ software development methodology: Extreme Programming or XP [16]. This methodology is based partly on common practices (and looks in some respect like Natural Software Engineering).

Goal: XP addresses the *risks of software development*: hidden schedule slips, changing requirements, fluctuating programming team, and increasing defect rate. The latter results from continued attempts to fix errors and enhance functionality. These attempts let the system's structure decay. It becomes increasingly difficult to change anything without adding more errors than can be fixed [17].

Method: Due to modern software technologies the cost of program modification does no longer increase exponentially over time. Changes in software can be undertaken throughout the production process. And these changes are entailing only modest cost. This makes it possible to start with small initial investments and a system design of utter simplicity. From this starting point the software evolves incrementally. The customer takes part in this process and is allowed to develop his requirements further. The system's structure undergoes adequate changes by frequent *refactoring* activities.

There is no specification. *Story cards* are used instead to fix scenarios and use cases thought out by the customer. And with the help of a dedicated tester the customer writes the *test cases*, story by story.

Central features of XP are:

1. *Test driven development:* There is no specification. If the tests run, you are done for the moment. “When you can’t think of any test to write that might break, you are completely done” [16, p. 45].
2. *Pair programming:* All production code is written by two programmers looking at one machine, one of them thinking more strategically.
3. *Collective Ownership:* All persons are continually changing their roles. Everybody takes responsibility for the whole of the system.
4. *Communication by code:* Following coding standards and striving for simplicity of design results in readable programs. Code is used to communicate clearly and concisely thus reducing the need of extra documentation.
5. *Iterations:* The software evolves iteratively. Each iteration consists of a series of interwoven activities: Exploration (eliciting requirements by means of story cards), planning (customers and programmers agree on a date by which the smallest, most valuable set of stories will be done), writing test cases, coding and testing.

Lifecycle model and classification. XP extends over the whole lifecycle, and the underlying lifecycle model differs considerably from the one of IEC 61508. There are four phases of a project, called *preproduction* (which is seen to be “an unnatural state for a system and should be gotten out of the way as quickly as possible” [16, p. 131]), *productionizing* (certifying the software is ready to go into production), *maintenance* (“the normal state of an XP project”, [16, p. 135]), and *death*. During the entire lifetime the software development is done by iterations. Only the pace of evolution differs from phase to phase.

Discussion. Root cause analysis underscores the importance of specifications and documentation to safety related systems. In XP specifications are not present and documentation is undervalued. Extreme programming does not conform to the lifecycle model of IEC 61508. And it is not meant to do so. Therefore the XP cannot be classified within the proposed scheme. Some elements of XP like *pair programming*, *collective ownership*, and *communication by code* should be examined more closely to find out their possible contribution to the development of safety related software.

Conclusions and Outlook

The proposed procedure identifies the most effective SRE methods to be applied during the various lifecycle phases. This is done by root cause analyses applied to the disasters of the past. The procedure has been applied to a couple of incidents. This resulted in a preliminary selection and classification of SRE methods. Further work has to be done to further substantiate the result and to get a more comprehensive table of methods and their allocation to the lifecycle phases.

Table 2 Classification of methods derived from causal analysis.
 ([B...] is shorthand of [1, part 7, annex B...].)

Lifecycle Phase		Method
1	<i>Specification</i> Functional Requirements, Safety Integrity Requirements	Hazards and Operability Analysis (HAZOP) [B34] Fault Tree Analysis (FTA) [B28] Event Tree Analysis (ETA) [B23]
2.1	<i>Realisation</i> Design	Modular approach: hierarchical decomposition, information hiding [B43], [13]
2.2	<i>Realisation</i> Implementation (Coding)	Modular approach: treating interface descriptions as specifications, use of mathematical and logical notation for outermost exactness, documentation [B43], [13]
2.3	<i>Realisation</i> Validation and Verification	Interface analysis: rigorous module check whether all requirements are met by the specification [3] Failure Mode and Effect Analysis (FMEA) [B26] Boundary value analysis [B4] Equivalence classes and input partition testing [B19]
3	<i>Installation and Commissioning</i>	Inspection and functional tests need to be specified as explicitly as practicable
4	<i>Operation and Maintenance</i>	Design for Maintenance [4, p. 27]
5	<i>Changes after Commissioning</i> Modification, Reuse and Retrofit	The same as during the realisation phase (design, implementation, validation and verification) Regression test Interface analysis

Bibliography + Links

- [1] IEC 61508: Functional Safety (deutsch: VDE 0801: Funktionale Sicherheit). Part 1-7 (1998, 2000)
- [2] Grams, T.: Putting the Normative Decision Model into Practice. In: Elzer/Kluwe/Boussoffara (eds.) "Human Error and System Design and Management", pp. 99-107. Lecture Notes in Control and Information Sciences 253. Springer, London 2000
- [3] Leveson, N. G.: Safeware. System Safety and Computers. Addison-Wesley, Reading, Massachusetts 1995
- [4] Health and Safety Executive (GB): Out of Control. Why control systems go wrong and how to prevent failure. HSE Books 1995
- [5] Grams, T.: Grundlagen des Qualitäts- und Risikomanagements. Zuverlässigkeit, Sicherheit, Bedienbarkeit. Vieweg Praxiswissen, Braunschweig, Wiesbaden 2001
- [6] Popper, K. R.: Conjectures and Refutations. The Growth of Scientific Knowledge. Routledge, London 1989
- [7] Neumann, P. G.: Computer Related Risks. The ACM Press 1995
- [8] National Transportation Safety Board: Marine Accident Report, Grounding of the Panamanian Passenger Ship Royal Majesty on Rose and Crown Shoal near Nant-

- cket, Massachusetts, June 10, 1995. Report Number NTSB/MAR-97/01 (<http://www.nts.gov>)
- [9] Husemann, P.; Ladkin, P. B.; Sanders, J.; Stuphorn, J.: A WBA of the Royal Majesty Accident. RVS Group, Faculty of Technology, University of Bielefeld, Draft Version of July 7, 2003 (Browse <http://www.rvs.uni-bielefeld.de/Bieleschweig/index.html>)
 - [10] Grams, T.: Operator Errors and their Causes. In: Computer Safety, Reliability and Security. Proceedings of the 17th International Conference, SAFECOMP '98, Heidelberg, Germany, October 1998 (W. Ehrenberger, Ed.). Lecture Notes in Computer Science. Springer, Berlin Heidelberg 1998 (pp. 89-99)
 - [11] Kletz, T.: An Engineer's View of Human Error. 3rd edition. Institute of Chemical Engineers 2001
 - [12] Leveson, N. G.: Software Safety: Why, What, and How. Computing Surveys 18 (June 1986) 2, 125-163
 - [13] Parnas, D. L.: The Secret History of Information Hiding. In: Broy, M.; Denert, E. (Eds.): Software Pioneers. Contributions to Software Engineering. Springer, Berlin, Heidelberg, New York 2002, pp. 399-409
 - [14] Lions, J. L.: ARIANE 5 - Flight 501 Failure. Report by the Inquiry Board. Paris, 19 July 1996. <http://www.esrin.esa.it/htdocs/tide/Press/Press96/ariane5rep.html>
 - [15] Agile Alliance: Manifesto for Agile Software Development. <http://www.agilemanifesto.org>
 - [16] Beck, K.: Extreme Programming Explained. Embrace Change. Addison-Wesley, Boston 2000
 - [17] Berry, D. M.: The Inevitable Pain of Software Development: Why There Is No Silver Bullet. University of Waterloo, Ontario, Canada (15 January 2003). <http://se.uwaterloo.ca/~dberry/>