Hojat Mohammadnazar

IMPROVING FAULT PREVENTION WITH PROACTIVE ROOT CAUSE ANALYSIS (PRORCA METHOD)

UNIVERSITY OF JYVÄSKYLÄ
DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS
2016

ABSTRACT

Mohammadnazar, Hojat
Improving fault prevention with proactive root cause analysis (PRORCA method)
Jyväskylä: University of Jyväskylä, 2016, 95 p.
Information Systems, Master's Thesis
Supervisor(s): Pulkkinen, Mirja
Measures taken to prevent faults from slipping through to operation can secure development of highly reliable software systems. One such measure is analyzing the root causes of reoccurring faults and preventing them from ever appearing again. PRORCA method was developed in order to provide a proactive, lightweight and flexible way for fault prevention. To this end, PRORCA method relies on expert knowledge of the development context and development practices to identify individuals' erratic behaviors that can contribute to faults slipping through to operation. The development of the method was done according to teachings of design science research. Three expert interviews with representatives of a case company supported the development of PRORCA. The first interview helped the problem identification and solution generation, while the other two interviews were carried out with the purpose of demonstrating the use of the PRORCA method in two different projects. Using the PRORCA proved to be easy and insightful findings were drawn from conducting it with respect to individuals' erratic behavior in each project. Proactive analysis of faults using the PRORCA method supports development of highly reliable software systems in a simple, flexible and resource-friendly manner.
Keywords: Software reliability, fault prevention, contextual factors, proactive root cause analysis

FIGURES

TABLES

TABLE OF CONTENTS

1 INTRODUCTION

With increasing presence of automated computation and networked communication, quality measures of systems responsible for delivering these services become critical. Quality attributes often discussed for such systems are dependability, and security (Avižienis, Laprie, Randell, & Landwehr, 2004). The former, dependability, encompasses several attributes one of which is reliability (Avižienis et al., 2004).

Reliability as the degree to which a system can continue to operate correctly in a specified duration of time has been a matter of concern in computer engineering literature and other related fields from the early ages of computer evolution (Goel, 1985). In the early days, the focus of research was on hardware reliability and performance (Goel, 1985). The focus, however, has shifted from hardware to software from 1970's onward as developers and users have come to realize that even though, unlike hardware, software is not subject to wear and tear, as a human activity, software development is not free of fault and malice (Avižienis et al., 2004; Goel, 1985).

The correctness of operation as a defining characteristic of reliability is faltered with occurrences of failures. Reliability of a system suffers with occurrences of service failures (Avižienis et al., 2004). Unsatisfactory reliability might have catastrophic consequences on the user(s) and the environment in safety-critical (Bishop, 2013) and business-critical (Børretzen, Stålhane, Lauritsen, & Myhrer, 2004) systems. Several instances of aircraft and spacecraft accidents due to software failures are presented in Favarò, Jackson, Saleh and Mavris (2013) and Leveson (2004), respectively. According to Lyu (2007), software reliability target in many projects is set as five 9's or six 9's which could be understood as 10^{-5} to 10^{-6} failures per execution hour. However, the threshold that distinguishes between high and low reliability is a matter of debate (Voas, & Miller, 1995). For example, Butler and Finelli (1993), claim that ultrahigh reliability needed by safety-critical applications is 10^{-7} to 10^{-9} failures for 1 to 10 hour missions.

Even though it is a common practice, setting a reliability target in terms of failures and quantitative assessment of software reliability is not recognized as an

absolutely justified way to achieve reliability (Butler, & Finelli, 1993; Littlewood, & Strigini, 1993). Stressing the differences between software and hardware, Butler and Finelli (1993), challenged the software reliability community to leave the prevalent idea of quantitative software reliability modeling and provide 'credible' methods for developing reliable software. To this end, some software standards, such as ECSS software dependability and safety standard (ECSS-Q-HB-80-03A 2012), do not advise setting numerical reliability targets in terms of failures and using reliability models. These standards assert that their rigorous design ensures high reliability upon compliance with the practices and procedures. Bishop (2013) reports that projects complying with IEC 61508 (2010) Level 4 will have a failure rate as low as 10^{-9} per hour.

Similar to the existential nature of the relationship between failures and deviation from correct operation, there is a relationship between faults and failures. A fault could be considered as a flaw in the software that can potentially lead to a failure (Avižienis et al., 2004). Consequently, it is possible to assume a cause-effect relationship between faults and reliability. However, caution is advised in drawing a direct cause-effect relationship between the number of faults and reliability (Fenton, & Neil, 1999). Fenton and Neil (1999) argued that drawing such a relationship necessitates a good understanding of the relationship between faults and failures which is still not available. Hamill and Goseva-Popstojanova (2009) addressed the complexity of the relationship between faults and failures and noted the possibility of one-to-many, many-to-one and many-to-many relationships between faults and failures. Adams (1984) demonstrated that a large number of failures are caused by a small number of faults. Nevertheless, the existence of the relationship between faults and failures and consequently faults and reliability is undeniable.

Lyu (1996) suggested that (1) fault prevention, (2) fault tolerance, (3) fault removal, and (4) fault forecasting are four technical areas that make development of highly reliable software possible. While explaining reliability as an attribute of dependability, Avižienis et al. (2004), presented the same means for development of highly dependable systems. Fault prevention calls for the elimination of the causes of the faults via process modifications, thus reducing the chances of fault introduction during development. Fault tolerance techniques are used to develop mechanisms into the software in order to avoid service failures in the presence of faults. Fault removal refers to techniques and practices that are utilized to reduce the number and severity of faults. Finally, Fault forecasting is estimating the present number, the future incidence, and the likely consequences of faults. (Lyu, 2007; Avižienis et al., 2004.) Therefore, it can be inferred that 'credible' methods for developing highly reliable software should be drawn from these four technical areas.

There is a tendency in the research community to undermine fault prevention (Alho, & Mattila, 2011). This tendency was criticized by Alho and Mattila (2011)

who described failing to care for prevention as 'shortsighted' and called for further research into fault prevention. Alho and Mattila (2011) argued that fault tolerance techniques cannot protect applications against all possible faults and prediction of unexpected faults can be expensive. Furthermore, fault forecasting research has yet to reach a consensus on the metrics with highest predictability (Catal, & Diri, 2009; Fenton, & Neil, 1999; Hall, Beecham, Bowes, Gray, & Counsell, 2012; Radjenović, Heričko, Torkar, & Živkovič, 2013).

At this point, it is necessary to note that there is a certain ambiguity in referring to fault prevention which needs clarifying. Prevention could potentially mean prevention of faults slipping-through to operation or preventing fault introduction during implementation. Avižienis et al. (2004), considered prevention as part of general engineering in which process modifications are made to reduce fault introduction during implementation. However, evidently, general engineering includes fault detection and fixing activities with the intention to produce high quality products. Additionally, software development processes are designed according to software development methods and standards, all of which mandate existence of testing and review processes. As a result, it could be postulated that fault prevention includes fault removal activities with the purpose of preventing faults from slipping through to operation. In this research, fault prevention and prevention of faults from slipping through to operation are used interchangeably.

Process improvement models such as CMMI (2010), ISO/IEC 12207 (2008), and Six Sigma are the prime candidates for delivering fault prevention. The effect of Process improvement, particularly those presented in CMMI, on reducing the number of faults slipping through to operation has been empirically approved. Notably, Diaz and Sligo (1997), stressed that, in their case organization, each CMM level upgrade in a project reduced the number of faults introduced to roughly half the number in previous levels. In a similar vein, Harter, Kemerer, and Slaughter (2012) reported significant reduction in likelihood of introducing severe faults in higher levels of CMM. The effect of Consistency in adopting CMM practices on introducing faults has also been the subject of studies. Krishnan and Kellner (1999) studied consistent adoption of CMM practices and demonstrated that such adoption is significantly associated with lower number of faults being introduced. Huang, Liu, Wang, and Li (2015) demonstrated that lower number of total faults, minor faults and severe faults slipping through to operation are achieved when adoption of CMM practices is done consistently.

One of the practices included in many software process improvement models is analysis of root causes of faults (Kalinowski, Travassos, & Card, 2008). For example, one of the key process areas of the CMMI level 5 is 'Causal Analysis and Resolution' (Shenvi, 2009). There are a myriad of methods in the literature, offering systematic ways to identify the root causes of faults (Chillarege, et al. 1992; Card, 1998; Grady, 1996; Kalinowski et al., 2008; Lehtinen, Mäntylä, & Vanhanen, 2011).

Whichever method is chosen, the goal is identification of the root causes of reoccurring faults and preventing them from being introduced in future projects or in the same project by resolving their root causes. Such methods are known by the names such as Root Cause Analysis (RCA), Defect Causal Analysis (DCA), and Common Cause analysis to name a few. RCA methods do drive process improvement but their merits are not limited to it. Most of the RCA methods rely on statistical analysis of fault data for identifying reoccurring faults. To make such statistical analysis, the fault data should be collected in a formulated manner. To this end, several fault classification schemes such as Orthogonal Defect Classification (Chillarege et al., 1992), and Defect Origins, Types, and Modes (Grady, 1996) have been proposed by researchers.

Even though reliance on fault data is insightful (Grady, 1996), it comes at a high price for RCA methods. Fault data is difficult to collect (Mohagheghi, Conradi, & Børretzen, 2006); and its collection needs upfront investment and personnel training (Carrozza, Pietrantuono, & Russo, 2015). These difficulties have rendered the majority of existing RCA methods resource intensive and inappropriate for small and medium-sized enterprises (SMEs) (Lehtinen et al., 2011). More importantly, existing RCA methods are reactive in nature. In a longitudinal study of software process improvement model implementation, Fitzgerald and O'Kane (1999) found out that the prevention activities championed by CMM are reactive in nature.

In this research, RCA as one of the key instruments available for fault prevention is brought into the spotlight, and a new proactive RCA (PRORCA) method is developed to address the difficulties of conducting RCA using existing methods. This research is a response to Alho's and Mattila's (2011) call for further research into fault prevention. For the purposes of the research, Design Science Research Methodology (DSRM) (Peffers, Tuunanen, Rothenberger, & Chatterjee, 2007) is adopted as a nominal process and mental model. The research is carried out in three phases which are mapped to stages of DSRM. The problem identification and demonstration stages of the DSRM which are mapped to phase one and phase three of this research, respectively, are supported by three qualitative interviews with representatives of a case company in the domain of avionics and embedded systems. Furthermore, a systematic mapping study (Kitchenham, & Charters, 2007) is performed in the first phase for problem identification and solution innovation. Moreover, directed content analysis (Hsieh, & Shannon, 2005) is performed in phase two on a collection of academic articles.

The PRORCA method has three steps, namely, context mapping, erratic behavior mapping and corrective action innovation. The main idea in PRORCA is proactive identification of individuals' erratic behaviors based on mismatches between development context and development practices. Preventing such erratic behaviors that can contribute to fault introduction, ineffective and inefficient fault detection and, ineffective and inefficient fault fix would, in return, prevent faults

from slipping through to operation. In the course of the research, taxonomy of contextual factors affecting fault slipping through to operation is developed using directed content analysis (Hsieh and Shannon 2005) on existing publications. The taxonomy is the key tool for identification of mismatches between the development context and practices.

The rest of this document is organized as follows. First, the relationship between errors, faults, and failures is outlined. A clear description of fault prevention is outlined in the third section. Next, RCA will be explained. In the fifth section the research approach is discussed. Three phases of the research are included in this section. Problem identification, and objectives and solution innovation are discussed in phase one. Design and development of taxonomy of contextual factors and PRORCA are included in phase two. And demonstration of the use of the PRORCA method and its evaluation are presented in phase three. This section is then followed by discussion, limitations and finally conclusion.

2 FAULTS, ERRORS AND FAILURES

Central to development of highly-reliable software systems through prevention is the relationship between faults, failures and errors. There exist two approaches to explain the relationship between errors, faults and failures. A reader must be vigilant with respect to which one of these approaches is taken when interpreting the results in the literature. The distinction between the approaches is drawn by the way errors are defined. In one approach errors are considered a wrong internal state of a software system, while in the other errors are considered a wrong-doing of a human that produces incorrect results.

Avižienis et al. (2004) is one of the advocates of the first approach. According to Avižienis et al. (2004), a failure is a deviation from correct service which occurs either when the specification is not complied with or when the specification is wrong. In case of a failure, a system's external state is incorrect. What precedes this incorrect external state is usually an incorrect internal state which is known as an error (Avižienis et al., 2004). It could be said that a failure occurs when an error reaches the system's interface (Hanmer, McBride, & Mendiratta, 2007). Faults are potential flaws and/or imperfections that if activated might lead to errors (Børretzen, & Dyre-Hansen, 2007). A fault might cause an error in the internal state of the system which does not affect the external state (Avižienis et al., 2004). The relationship between faults, errors and failures in this approach is shown in FIGURE 1.

The second approach is advocated by ISO/IEC 24765 (2010) standard. In this approach an error is a wrong-doing of a human that produces incorrect results. A fault, then, is a manifestation of an error which could possibly lead to a failure. Alternatively, an error could be a wrong step, process or data definition that manifests itself as a fault (ISO/IEC 24765 2010). A software failure, then, is

"termination of the ability of a product to perform a required function or its inability to perform within previously specified limits" (ISO/IEC 25000 2005).

Regardless of the approach, a failure can occur due to a fault. In both approaches a fault can exist both in executable code and documents including specification and requirement. Fault introduction can occur at any stage during the development process. An introduced fault might propagate to subsequent phases (Van Moll, Jacobs, Kusters, & Trienekens, 2004). Additionally, in both approaches it is emphasized that a fault might not necessarily cause a failure. Alternatively, a failure might be due to several faults activated simultaneously. Another possibility is that a fault remains dormant during the whole lifetime of a system without ever causing any failures. Hammil and Goseva-Popstojanova (2009) noted the possibility of one-to-many, many-to-one and many-to-many relationships between faults and failures. In other words, there is a complex relationship between faults and failures all aspects of which are not exactly known (Fenton, & Neil, 1999).

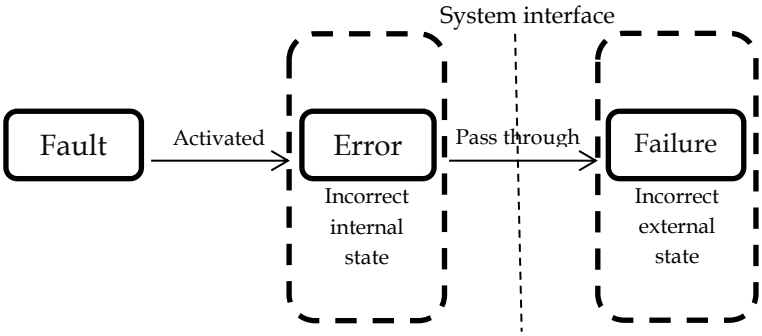

FIGURE 1 Fault and failure relationship adopted from Avižienis et al. 2004

It is important to note that these approaches and the definitions provided are not always adopted by different researchers as they are represented here. Moreover, the definitions, particularly those in software standards, have been subject to change over the years. For example, Boehm, Mcclean and Urfrig (1975) used the term error to refer to what was described as fault above; a flaw that can lead to a failure. Basili and Rombach (1987) adopt the second approach; however, they adopt the definitions in IEEE-Std-729 (1983) which might have minor differences with ISO/IEC 24765 (2010). Plus, the terms "fault", "defect" and "bug" are very often used interchangeably. Exceptions exist though. For instance, IEEE-Std-1044 (2009) differentiates between defects and faults. Consequently, interpretation of the pervious discussions and findings in the literature must be done with careful attention.

In this research, errors are left out and when necessary to refer to the cause of faults, the term root cause is deployed. Since faults and failures are defined almost identically in both approaches, they are adopted as was explained in this section. The terms fault and defect will be used interchangeably as well.

An example that can represent the fault and failure sequence is provided by Favaro et al. (2013) in which a failure in an aircraft control software led to uncontrolled maneuvers of a 777 Boeing aircraft. In this scenario, the aircraft boarded with one failed accelerometer (#5) out of six. Such a failure was predicted in the software requirements. In such a case the software was designed not to consider the data coming from the failed accelerometer. However, when, in an unpredicted event another accelerometer failed (#6) after engagement of auto-pilot, a fault in the design of the control software was activated. This fault allowed the data from accelerometer #5 to be included in calculation of acceleration values. This failure of the software to comply with specification led to sudden uncontrolled maneuvers of a 777 Boeing aircraft. Fortunately, this incident did not lead to any casualties or physical damage.

3 FAULT PREVENTION

Avižienis et al. (2004) stated that 'fault avoidance' as a combination of 'fault prevention' and 'fault removal' is a way to aim for development of systems that are free from faults. It is noteworthy that 'fault prevention' from the viewpoint of Avižienis et al. (2004) is one of the raison d'etre of development methods which reduces the number of faults introduced during development. This conception of fault prevention is limited to reducing 'fault introduction' during development. Bearing in mind that Avižienis et al. (2004) conceptualized 'fault removal' as both 'fault detection' and 'fault fixing', it can be postulated that (1) reducing fault introduction during development, (2) fault detection and (3) fault fixing can help to 'avoid faults'. In other words, fault avoidance is preventing faults from slipping through to operation. However, preventing faults from slipping through to operation is essentially the same as 'fault prevention'. From this perspective, fault prevention is a larger system in which the goal is to prevent faults from slipping through to operation. This larger system is what Avižienis et al. (2004) called 'fault avoidance'. However, since the conception of 'fault prevention' as prevention of faults from slipping through to operation satisfies the needs of this research and since adding a new term to the already dense and dark terminology jungle of dependability and reliability research is not on this research's agenda, the term 'fault avoidance' will not be used. Instead 'fault prevention is used to refer to (1) reducing fault introduction during development, (2) fault detection and (3) fault fixing.

It follows, based on this new conception of fault prevention that (1) fault introduction during development, (2) ineffective and inefficient fault detection and (3) ineffective and inefficient fault fixing are contributing elements to faults slipping through to operation. FIGURE 2 depicts the contributing elements to faults slipping through to operation. FIGURE 2 is not a process model and is not intended to show a sequence. The connectors in this model show a causal effect and the model itself is a causal one. For example, inefficient and ineffective fix can

lead to fault introduction. Alternatively, it can lead to faults slipping through to operation.

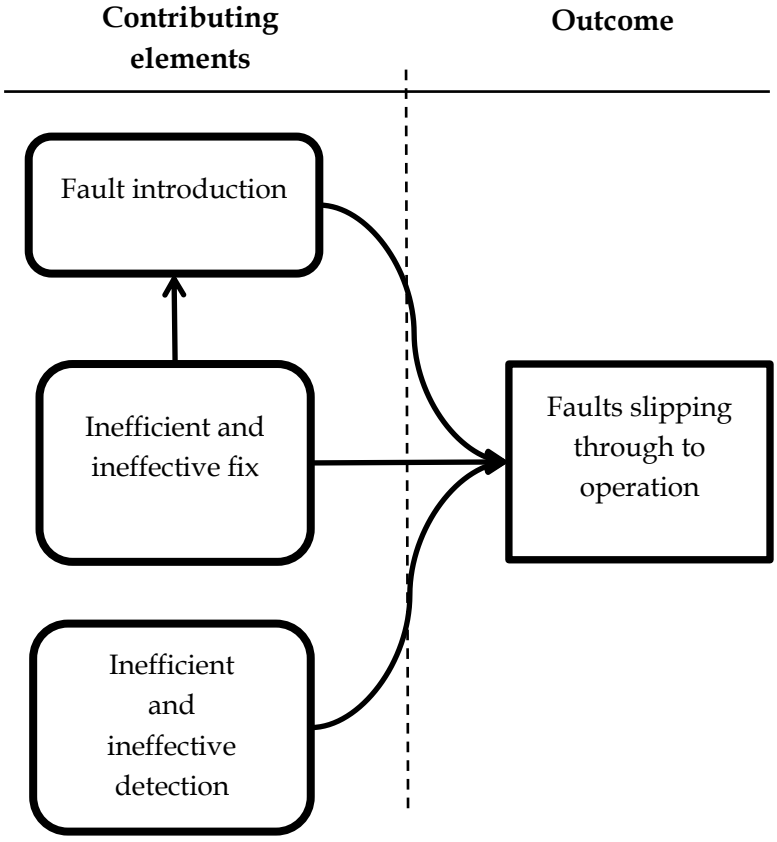

FIGURE 2 Fault prevention model

It is self-evident that unless a fault introduced during development is effectively and efficiently detected and fixed, it slips through to operation. Effectiveness of detection cannot be undermined. Ineffective detection, delivered by inappropriate testing and review practices, means that an introduced fault can go unnoticed and, eventually, slip through to operation. Effective fault detection mandates sufficient fault detection activities. As a matter of fact, one of the applications of software reliability growth models has always been notifying managers that enough fault detection has taken place to secure reliable operation of software (Butler, & Finelli, 1993; Carman, Dolinsky, Lyu, & Yu, 1995; Goel, 1985).

However, effectiveness is not all there is to fault detection; the efficiency of detection is also a matter of concern. According to the infinite monkey theorem, if a monkey is given infinite amount of time hitting keys randomly on a typewriter, it will eventually input a legible text. Similarly, if testers are given infinite testing time, they will eventually find all the faults in a piece of software. The same can be

argued for reviews. This is not, however, practical in today's turbulent and dynamic business environment. Testers can dedicate only a limited amount of time to detection activities and reviews do not span more than a few hours. In fact, Butler and Finelli (1993) argued that achieving ultrahigh reliability is not practical because it would require "testing beyond what is practical". It comes as no surprise, then, that inefficient defect detection could lead to faults going unnoticed during defect detection and slipping through to operation.

As much as fault detection is valuable, it is not enough to prove that if detected, a fault, is prevented from slipping through to operation; a fault needs to be fixed effectively and efficiently. If a fault is not fixed in time or with acceptable quality it may very well slip through to operation. A bad fix, on the other hand, could introduce additional faults (Christenson, & Huang, 1996; Whittaker, 2000). Whittaker (2000) emphasized the possibility that even though a bad fix could remove the original fault, still it can introduce new faults. Alternatively, a bad fix might introduce new faults without actually fixing the original fault (Whittaker 2000). Additionally, several authors including Li, Sun, Leung, and Zhang (2013), Kim, Zimmermann, Pan, and Whitehead (2006), and Canfora and Cerulo (2005) have indicated that fault-fixing changes can introduce further faults.

Lack of attention to any of the aforementioned activities can contribute to faults slipping through to operation. This contribution, depicted in FIGURE 2 could eventually lead to software failures and poor software reliability. These contributing elements are well-known and have been under investigation in software quality research before. Jacobs, Van Moll, Kusters, Trienekens, and Brombacher (2007) studied influential factors leading to defect introduction and defect detection. According to Jacobs et al. (2007) "the injection of defects should be minimized and the detection of defects should be maximized". Furthermore, implementation, testing and fixing activities were recognized as key improvement points for software quality improvement in Carrozza et al. (2014). These authors performed a defect analysis study in order to find effectiveness and efficiency bottlenecks during implementation, testing and fixing activities.

4 ROOT CAUSE ANALYSIS

Root cause analysis (RCA) is considered to be a key instrument to defect prevention and process improvement. RCA is a structured investigation to identify the underlying causes of faults. RCA can be performed both during the development and after product release. In the former case, RCA can result in in-process improvements (Chillarege et al., 1992) while in the latter, it helps create an organizational portfolio by which lessons learned from one project can be put into practice in later projects (Leszak, Perry, & Stoll, 2002).

Lehtinen et al. (2011) identified three common steps to all RCA methods - target problem detection, root cause detection and corrective action innovation. The general idea behind RCA is to identify patterns that reoccur with respect to faults, identify the root causes, and provide improvement suggestions.

Two forms of RCA have been reported in the literature: Qualitative RCA and Quantitative RCA. Qualitative RCA is an effective but resource-intensive process whereby root causes of faults are analyzed one-by-one by a team of experts (Grady, 1996; Mays, Jones, Holloway, & Studinski, 1990). Reliance of this form of RCA on human capabilities and high cost of implementation is considered as its downsides (Chillarege et al., 1992). However, recently, ARCA method was proposed by Lehtinen et al. (2011) as a lightweight approach to qualitative RCA. This approach is different from the other qualitative methods in that, even though, it is done qualitatively, it only relies on qualitative methods such as focus group meetings for target problem identification. Not all faults are analyzed in the ARCA method (Lehtinen et al., 2011); only the ones that a group of experts identify via a systematic approach. Such an approach is supposed to make RCA more applicable in SMEs that are often reluctant to conduct a resource-intensive analysis (Lehtinen et al., 2011).

Quantitative RCA is guided by statistical fault data analysis in problem identification stage. Statistical fault data analysis most often relies on data collected via fault reports. Fault reporting is formalized via fault classification schemes. In

quantitative RCA, statistical methods are utilized to visualize patterns that might reflect issues in development process. The root causes of such issues are then identified. There are a myriad of methods for identifying the root causes. Most famous among them is creating a fishbone diagram (Kalinowski et al., 2008) to record cause-effect relationships. Lehtinen et al. (2001) reported cases where fault tree diagrams, causal maps, matrix diagrams, scatter charts, logic trees, and a causal factor charts were used. Unfortunately, not much has been said on corrective action innovation (Lehtinen et al., 2011). Corrective actions are reported to be derived using qualitative approaches such as brainstorming, brainwriting, interviews, and focus group meetings (Card, 1998; Kalinowski et al., 2008; Lehtinen et al., 2011).

Conducting quantitative RCA is tightly coupled with the fault classification scheme of choice. The main function of a fault classification scheme is to determine a minimum set of attributes that allow slicing the fault data in various ways to provide visibility into problematic areas in the software development process (Bridge, & Miller, 1998). There are numerous classification schemes in the literature. The most well-known are Orthogonal Defect Classification (ODC) proposed by Chillarege et al. (1992) and developed at IBM, the Defect Origins, Types, and Modes scheme developed in HP, also known as the HP scheme (Grady, 1996), and the scheme proposed in IEEE Std. 1044. Other known schemes include those presented by Binder (2000) and Beizer (1990).

In order to ease the selection of a classification scheme, researchers have made efforts to evaluate them against each other. Huber (2000) compared ODC and HP schemes across five dimensions- Data, Process, Specification/Requirements, Defect Type, and Resource. Vallespir, Grazioli, and Herbert (2009) proposed a framework for evaluating fault classification schemes and compared the aforementioned schemes. Their comparison revealed that fault type is included in all fault classification schemes. Furthermore, Kalinowski et al. (2008) found two types of information being addressed by the fault classification schemes they reviewed, namely, fault information to be collected and fault types.

The HP scheme defines three high-level attributes and provides a set of possible values for each attribute. These attributes are Origin, Type and Mode (Grady, 1996). Depending on what Origin value is selected for a fault, values available for the Type attribute differ. On the other hand, the underlying idea in ODC (Chillarege et al., 1992) is that defect data should be collected in a way that allows classes of defects to be associated with stages of development process (Chillarege et al., 1992). Orthogonality refers to the independence of value of each attribute from the values of the other attributes (Vallespir et al., 2009). ODC calls for collection of at least two attributes with utmost importance - Defect Type and Defect Trigger. Six other attributes, namely, impact, target, activity, qualifier, source, and age are also recommended to be collected but they are supporting attributes and their collection is not of existential importance to ODC. Fault

classification in IEEE std. 1044 is similar to ODC in structure (Mellegård, & Torner, 2012). Among ODC, IEEE std. 1044 and HP scheme, collection of severity is only addressed in IEEE std. 1044 classification, while mutually exclusive attribute values are addressed in all (Vallespir et al., 2009). Mutually exclusive attribute value means that if one value is selected for an attribute no other values can be selected (Vallespir et al., 2009).

In practice, it is hard to believe that the well-known schemes are adopted fully and completely. Card (2005), for example, stated that classification of fault types should support analysis of fault data based on specific objectives of organizations. Fault classification schemes need some tailoring to fit the different needs and objectives of organizations (Mellegård, & Torner, 2012). Examples of customized classification schemes exist in the literature. Both El Emam and Wieczorek (1998) and Lutz and Mikulski (2004) customized ODC to fit their goals. Freimut, Denger and Ketterer (2005) developed a customized classification scheme based on ODC and HP schemes. Raninen, Toroi, Vainio and Ahonen (2012) introduced their own customized classification scheme based already existing schemes. Mellegård, and Torner (2012) tailored the IEEE std. 1044 classification to be used in a company in automotive industry. Leszak et al. (2002) developed a classification scheme and compromised the mutual-exclusivity of cause attribute value. According to Leszak et al. (2002) a fault might have several causes or no causes at all. Freimut et al. (2005) presented an approach for developing and evaluating customized classification schemes.

With all the alternative RCA methods available, Software development companies should adopt one that fits their goals and resources best. If there are limited resources available for RCA, the decision to adopt or customize one of the well-known fault classification schemes and perform quantitative RCA should be studied beforehand with due attention to its downsides. Such a decision can add overhead to developers' work and if not fully complied with, might not be as effective as expected.

5 Research approach

Software reliability and its related topics have historically been researched in the software engineering community. Even though exceptions exist of research being published in other fields such as information system (Zahedi, 1987), the predominant approach has been following guidelines of software engineering research.

A research effort intended for development of a solution to an engineering problem could benefit from a framework that formalizes conducting, validating and reporting the research (Kitchenham et al., 2002). For such a framework to prove valuable a number of requirements should be satisfied. Such a framework should:

1) promote theory building
2) have clear principles and rules
3) entail a clear process for carrying out research

Theory is the basic means for communicating knowledge (Sjøberg, Dybå, Anda, & Hannay, 2008). It sets the foundations on which a sound solution can be developed and communicated, hence, the first requirement. Declaring clear principles and setting proper rules is an existential feature of a research framework. Principles and rules often presented as guidelines that assist the researcher to make the right decisions, avoid pitfalls and communicate correctly (Kitchenham et al., 2002). However, to achieve this goal the principles and rules should be complemented with a clear research process. A clear research process provides an optimal roadmap that guides the researcher, from design, and delivery to communication and evaluation of a solution. Accompanied by guidelines and theory, such a roadmap promises a way to arrive at a robust and rigorous solution.

Software engineering research is reluctant to build and adopt theories (Hannay, Sjøberg, & Dybå, 2007). Even though guidelines do exist in software

engineering literature, they are either too abstract or too detailed (Kitchenham et al., 2002) or they lack a defined process. Plus, according to Kitchenham et al. (2002), the level of the standards in conducting empirical software engineering and subsequent meta-analysis of software engineering studies is low. Design Science research (DSR), on the other hand, as an alternative, while rooted in engineering (Hevner, March, Park, & Ram, 2004; Peffers et al., 2007), showed signs of satisfying the three requirements set above.

DSR is one side of the coin in information system research (Hevner et al., 2004). While the behavioral science research in information system research examines behaviors and attitudes related to a business need, DSR focuses on utility and provides pragmatic solutions in the form of artifacts to satisfy that business need. The focus in DSR is essentially on developing an artifact. An artifact could be a set of constructs, models, methods or an instantiation of a system (Hevner et al., 2004). Design science research emphasizes adoption and building of theories and stresses the importance of prior knowledge base (Hever et al., 2004; Peffers et al., 2007). Plus, as argued by Walls, Widmeyer and El Sawy (1992), DSR entails both a product aspect and a process aspect. The features of DSR provided evidence that it can provide a suitable framework that satisfies the three requirements set for carrying out this research.

DSRM proposed by Peffers et al. (2007) satisfies all of the requirements described above. As a result DSRM is selected as a framework of reference and mental model to guide the researcher in this research endeavor. DSRM (Peffers et al., 2007) is comprised of six stages, namely, (1) problem identification and motivation, (2) definition of objectives and solution, (3) design and development of an artifact, (4) demonstration, (5) evaluation, and (6) communication.

This research is carried out in three phases. As depicted in FIGURE 3, in each phase, one or two of the stages of DSRM are completed. In phase one, the researcher sets out to identify problems in fault prevention that would lead to lower software reliability. From the identified problems a solution is inferred and objectives are set. At the end of phase one, a model of elements and actors contributing to faults slipping through to operation is developed. Faults slipping through to operation are the main phenomenon that needs to be addressed in fault prevention. During the second phase, which is a one-to-one mapping of stage three in DSRM, taxonomy of contextual factors affecting faults slipping through to operation is developed based on the model presented at the end of phase 1. In the same phase, a proactive RCA (PRORCA) method is developed as a solution. The taxonomy of contextual factors is used as the underlying tool in conducting PRORCA. The third phase is a demonstration of using PRORCA in two small projects in a case company. A small project refers to a project that "a single individual can encompass and resolve any and all of the significant macro and micro issues involved in developing the system" (Boehm, 1975). An evaluation of applying PRORCA in these projects is done in phase three as well.

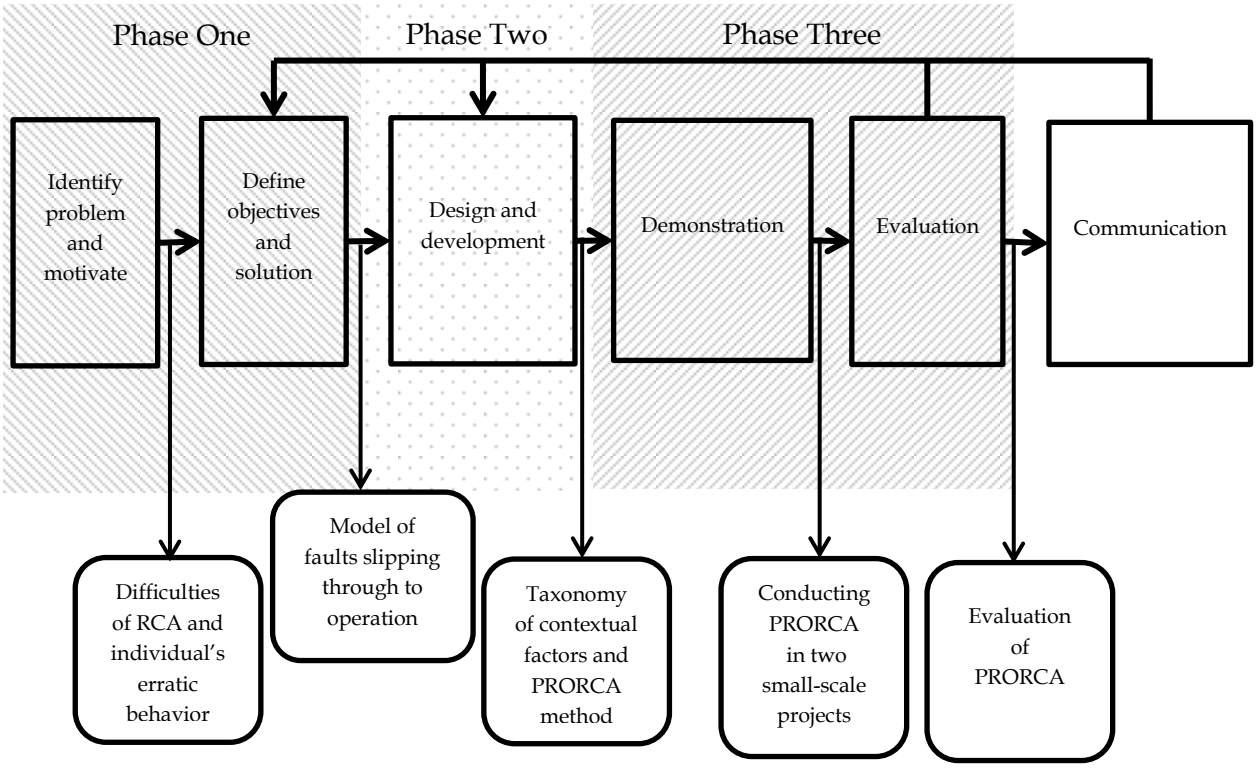

FIGURE 3 Research phases mapped to DSRM (Peffers et al., 2007) stages

Even though each phase of this study relies on a specific set of data, there are commonalities between all the phases which help form the cohesive whole of the research. The sources used for collecting data are interviews and published articles. The interviewees are representatives of a case company which is an international, privately-owned SME operating in avionics and embedded systems industry. They provide engineering services for their customers, mainly the European Space Agency (ESA), at different centers. Further detail on the research methods used in each phase is provided in the following sections in which each phase is discussed.

5.1 Phase one

The first phase of the research entails stages one and two in DSRM in which firstly the problems are identified and then further objectives and solutions are inferred from the identified problems. It is important to note that according to Peffers et al. (2007) entry point into the DSRM could be different in every research effort. In this research, the entry point to DSRM was stage one. In order to complete phase one, firstly, a review of the literature on software reliability and fault prevention were

carried out. Further, an interview with a representative of the case company was conducted.

The decision of conducting both literature review and interview in the first phase is well-justified. Since in stage one of DSRM the goal is identifying the problem, a review provided a wide variety of topics each addressing different problems in software reliability. This allowed capturing a big picture of reliability research and acquiring knowledge about underlying common problems in fault prevention. Kitchenham and Charters (2007), recommended that in cases where the scope of the topic is very wide, a systematic mapping study be conducted. By providing an overview of a topic, a systematic mapping study, establishes the research evidence in a topic (Kitchenham, & Charters, 2007). A systematic literature review is a stand-alone study that synthesizes the material in the literature (Okoli, & Schabram, 2010). Since the subject area of software reliability and fault prevention is wide and varied in scope, a decision was made to conduct a systematic mapping study rather than a comprehensive literature review.

Even though mapping studies and literature review studies are essentially different in their goals and comprehensiveness, their differences do not expand to guidelines. The systematic approaches recommended by Kitchenham and Charters (2007), and Okoli and Schabram (2010) both require a defined protocol for material extraction, material inclusion, and material exclusion.

For the purpose of material extraction, Webster and Watson (2002), recommended a three staged approach which starts with a keyword search and is later complemented by backward and forward reviewing of the citations. Kitchenham and Charters (2007), and Levy and Ellis (2006) made similar recommendations. Levy and Ellis (2006) emphasized the prominence of the studies in the initial set, though. Having a stopping rule for extracting new material is also of utter importance (Okoli, & Schabram, 2010). Following these guidelines, in this study, the following approach was taken:

1) Initial set generation: The initial set of papers was extracted from google scholar database using keyword search. The keywords were ‘software reliability, ‘fault prevention’, ‘software reliability engineering’. TABLE 1 shows the initial set.
2) Backward search: the references that were found relevant or that revealed important information were reviewed.
3) Forward search: using the ‘cited by’ feature of google scholar database relevant papers were identified and reviewed.
4) As the research unfolded new keywords, forward and backward search were complemented with further keyword search.
5) The stopping rule for extracting material was increasing frequency of repeating and irrelevant entries in backward and forward search. However, later on a calendar date constraint was also set to stop the backward and forward search.

In the beginning, the review was driven by the question: 'what are fault prevention techniques and capabilities recommended in the literature?' As the review was extended, it became clear that the problem was not that the techniques and capabilities were not known, but that they were not practiced or complied with. As the reasons for such behavior began to surface phase one began to take form.

Inclusion and exclusion in all the steps presented above was done based on the researcher's knowledge of the area. For inclusion of a study, first the title was investigated, if the title revealed new or relevant information regarding software reliability, fault prevention, fault detection, and RCA, that study was selected for abstract review. If the same conditions proved right for the abstract then the reference was included. Furthermore, if a study was considered to be seminal work in the field, it was included. Naturally, the exclusion occurred when a paper was not included for abstract review, or complete review.

TABLE 1 Initial set of academic articles

#	source
1	Carpenter, & Dagnino, 2014
2	Babu, Kumar, & Murali, 2012
3	Alho, & mattila, 2011
4	Hammil, & Goseva-Popstojanova, 2009
5	Lyu, 2007
6	Zelkowitz, & Rus, 2004
7	Dunn, 2004
8	Hermann, & Peercy, 1999
9	Musa, 1996
10	Leveson, & Turner, 1993
11	Zahedi, 1987
12	Goel, 1985
13	Børretzen, 2007

At the end of the review process a total of 168 academic articles and one PhD dissertation were reviewed. After analysis of the subject matter, these reviewed publications were categorized into 13 topic areas. TABLE 2 shows the topic areas that were covered and the number of articles reviewed in each one. The table indicates that most articles reviewed were in the 'fault detection', 'fault reporting and RCA', 'fault prediction' and 'fault reduction' topic areas. A complete list of reviewed articles is presented in appendix 6. Appendix 6 is organized in accordance with the topic areas represented in TABLE 2.

TABLE 2 Topic areas reviewed

Topic area	number of articles reviewed
fault detection	45
human factors	5
reliability modeling	7
Fault reporting and RCA	31
agile	9
fault prediction (change analysis,)	24
safety	7
maintenance	2
defect analysis	7
fault reduction	18
process improvement	3
tools	5
software reliability engineering	6

It is acknowledged that the approach taken for material extraction is vulnerable to lack of reliability, because its coverage of literature relies heavily on the initial set. In such a case, if the initial set is not well-chosen, the chance of missing important areas of research and seminal articles grows. Plus, this strategy for material extraction tends toward backward research rather than forward research (Jalali, & Wohlin, 2012). These problems were handled by conducting an expert interview. The interviewee was the leader of a team of four developers in the case company with years of experience as software engineer and system engineer in the avionics and aerospace industry. The prime function of the team under his leadership was research and development which in certain instances included safety-critical software development.

The benefits of the interview were threefold. Firstly, based on the interviewee's responses new research paths were investigated. Secondly, the interview increased the confidence of the researcher about the nature of the problem that was found and confirmed some of the problems recorded in literature. For example, it was after the analyzing the interview data that the mismatch between contextual factors and development practices came to light as an improvement opportunity. Plus, the interviewee pointed out the problems in fault reporting and reluctance to perform RCA. Lastly, the input provided by the interviewee prevented the researcher from going deep into research directions that had little value. For instance, the decision to abandon the topic of reliability modeling was founded on the responses of the interviewee.

All in all, the combination of the reactive nature of RCA techniques and the difficulties in its execution came to light as problems that can impede effective fault prevention. These problems coupled with the discovery that focusing on mismatches between the development context and the development practices is an

effective way to prevent individuals' erratic behavior, uncovered a solution for improving fault prevention. The solution is a proactive RCA technique that relies on identifying the mismatches between the development context and development practices to prevent faults from slipping through to operation. The difficulties in executing RCA, individual's erratic behavior and objectives and solutions are discussed in the following three sub-sections.

5.1.1 RCA difficulties

Despite highlighting the significance of proactive rather than reactive prevention of faults (Grady, 1996) the RCA literature has fallen short of instrumenting a shift from reactive approaches to proactive ones. RCA methods are still essentially reactive in analyzing root causes of faults and introducing countermeasures. RCA, as discussed in the literature, can be performed both during the development (in-process) and after product release (retrospective). Among the 31 academic articles reviewed in the topic area 'fault reporting and RCA', 17 of them, either presented an RCA method or carried out RCA in practice. TABLE 3 demonstrates that while 9 studies delivered retrospective RCA in practice, only 4 conducted in-process RCA. Two studies conducted RCA both retrospectively and in-process. There is an inconsistency between 8 studies that openly advocated in-process RCA and the number of in-process studies carried out. Lack of in-process studies is not very surprising though. It can be explained by reluctance of software companies to share sensitive fault data about their ongoing projects. Such fault information is the necessary requirement for carrying out RCA in all but one of the studies reviewed.

Retrospective RCA is openly reactive, thus the time of conducting RCA is not a matter of concern. It is championed to create an organizational portfolio by which lessons learned from one project can be put into practice in later projects (Leszak et al., 2002). In today's turbulent business environment where each project is different in nature and execution, however, the advantage gained by performing retrospective RCA is a matter of debate. It is arguable that the benefit of retrospective RCA is maximized in release-based projects in which RCA on past releases can provide improvement suggestions for future releases (Yu, 1998).

Meanwhile, the advocates of the in-process method claim that their approach would result in improvements and eventually fault prevention while the project is still under way (Chillarege et al., 1992). The question that begs to be answered then is when the RCA should be performed for in-process improvements to be delivered. TABLE 3 shows that among the papers advocating in-process RCA, only three explicitly recommended a time to perform RCA. Closer inspection of the recommended time reveals the reactive nature of in-process RCA. If RCA is supposed to be delivered at the end of each development stage then not much can

be done regarding the completed stages in an ongoing project. In case it is done after each iteration, in an iterative development project, yet again, the outcome of RCA will be useful in future iterations. The benefit of such an approach is maximized in complex projects in which several teams are working concurrently on a product but at different stages of development.

Furthermore, Lehtinen et al. (2011) identified three common steps in RCA methods, namely, (1) target problem detection, (2) root cause detection and (3) corrective action innovation. These steps imply the underlying assumption that a problem already exists, root causes of which should be identified. This assumption reveals the reactive nature of RCA methods as well.

TABLE 3 RCA approaches and timing

Source	**Approach recommended**	**Approach taken**	**Timing**
Basili, & Rombach, 1987	Retrospective	Retrospective	-
Bridge, & Miller, 1998	NA	Retrospective	-
Chillarege, et al., 1992	In-process	Retrospective	NA
Freimut et al., 2005	In-process	In-process	NA
Hayes Raphael, Holbrook, & Pruett, 2006	NA	Retrospective	-
Li, Li, & Sun, 2010	NA	Retrospective and In-process	NA
Bhandari et al., 1993	In-process	In-process	After each phase
Grady, 1996	Retrospective and In-process	NA	NA
Hong, Xie, & Shanmugan, 1999	NA	In-process	NA
Kalinowski et al., 2008	In-process	NA	right after each of phases or within a phase in exceptional cases
Lehtinen et al., 2011	In-process	Retrospective and In-process	NA
Leszak et al., 2002	In-process	Retrospective	NA
Lutz, & Mikulski, 2004	NA	Retrospective	-
Shenvi, 2009	In-process	Retrospective	NA
Yu, 1998	NA	Retrospective	-
Jalote, & Agrawal, 2005	In-process	In-process	After each iteration
Raninen et al., 2012	NA	Retrospective	-

In addition to the reactive nature, there are many reports of the difficulty of conducting RCA. First and foremost, reliance of RCA techniques on fault data, except that of Lehtinen et al. (2011), makes them vulnerable to fault reporting mechanisms of organizations. According to Børretzen et al. (2007), in practice, fault

reports are usually collected just for removal and unfortunately are not further analyzed to gain process improvement insights. Plus, fault reports that are collected in organizations usually have comprehensibility and inaccuracy issues (Børretzen, & Dyre-Hansen, 2007). Mohagheghi et al. (2006) have identified a number of problems in fault reporting processes. They reported ambiguous problem report fields as a source of confusion for developers. Definitions and terms might mean different things to different groups of stakeholders. Lack of attention to product releases, changes in report fields between releases, coarse-grained information in reports, and different report formats and reporting tools are other issues that these researchers witnessed in fault reporting practices of organizations (Mohagheghi et al., 2006). Lehtinen et al. (2011) argued that reliance of RCA on fault reports imposes a considerable amount of upfront investment, i.e. defect classification scheme definitions, procedure setup, establishment of data collection mechanisms, and personnel training. As a consequence, even though it might be effective for larger companies that have defined and strict processes, RCA methods relying on fault data might not be favorable in SMEs. Raninen et al. (2012) shared a similar view and claimed that fault reports are not efficiently analyzed in smaller software companies. Furthermore, non-immediate visible gains, required customization, change in people's routines (Carrozza et al., 2015), and impractical assumption of full knowledge of defects (Mellegård, & Torner, 2012) are other factors that make performing RCA difficult.

In this research, the interviewee's responses confirmed that fault data collection in small-scale development is not an institutionalized activity. In this case, the interviewee clarified his experience from the case company and other companies. Furthermore, the interviewee mentioned that no formal analysis of faults data is performed in the case company. He stressed the importance of being proactive when a fault trend is observed, though.

> What I have seen it the past is quite informal so you start collecting the fault data when your customer says 'hey what's going on' or when you have to report to your customer. But I have usually worked in very small teams, maybe at most three developers; in such a small scale development it tends not to be done in my experience. (interview 1)

> It's just common sense. If you see there are many faults in one part of the software maybe it's time to really put some more effort and to be more proactive in solving problems but I have not seen anything formal. (interview 1)

In order to liberate organizations from collecting fault data, Lehtinen et al. (2011) proposed the ARCA method. Instead of relying on fault reports, in the ARCA method (Lehtinen et al., 2011), identification of the problem is done by relying on knowledge of participants in focus group meetings. This approach has the benefit of being lightweight and is not prone to vulnerabilities of fault reporting. Identifying problems based on the knowledge of participants in focus group meetings has the benefit of identifying potential future problems, thus, liberating

the RCA from being reactive. Because of this characteristic, even though, proactive RCA was not addressed by Lehtinen et al. (2011) explicitly, it is arguable that the ARCA method comes closest to proactive fault prevention. Therefore, relying on the knowledge of participants rather than fault data was considered as the solution to proactive prevention of faults from slipping through to operation in this research.

5.1.2 Individual's erratic behavior

Investigation into the common root causes of faults in the literature revealed evidence that matching the development context and the development practices is a promising way to prevent individuals' erratic behavior, thus, preventing faults slipping through to operation. TABLE 4 shows several studies that have provided the academic community with categorizations of the common root causes of faults. These categories, in fact, exhibit their creators' implicit and explicit beliefs regarding the common root causes of faults.

TABLE 4 Root cause categories

Source	Developed artifact	Root cause category
Boehm, 1975	Taxonomy of software error Causes	Consistency, completeness, communication, clerical
Basili, & Rombach, 1987	Root cause scheme	Application errors, Problem-Solution errors, Semantics error, syntax error, Environment errors, Information Management errors and Clerical errors
Leszak et al., 2002	Classes of root cause	Phase-related, Human-related, Project-related, review-Related
Kalinowski et al., 2008	Most cited cause-categories in the literature	tools, input, people, and methods
Hayes, et al., 2006	Requirements common causes	noncompliant process, lack of understanding, human error
Walia, & Carver, 2013	requirement error taxonomy	people errors, process errors, documentation errors
Huang, Liu, & Huang, 2012	Root cause taxonomy for software defects	Human error, process error, tool problems, task problems

In order to investigate how faults are delivered, the distinct root cause categories identified in the literature are extracted from TABLE 4 and presented in column one of TABLE 5. Based on the definitions provided for each category of root causes, the actors who can deliver faults were identified. Column three of TABLE 5 shows the actor who can deliver faults caused by each distinct category of root

causes. As it is clear from TABLE 5, tools, individuals and processes are those who deliver the faults.

TABLE 5 Actors delivering faults in each distinct root cause category

Distinct root cause category	**Description**	**Actor**
Consistency	"The requirements were well understood, but conceptual errors were made in implementing them at the next stage" Boehm (1975)	Individual
Completeness	"There was an incomplete grasp of the requirements expressed or implicit in the previous stage." Boehm (1975)	Individual
Communication	"There was a misunderstanding of the requirements expressed in the previous stage." Boehm (1975)	Individual
Application errors	"due to a misunderstanding of the application or problem domain" Basili & Rombach (1987)	Individual
Problem-Solution errors	"due to not knowing, misunderstanding, or misuse of problem solution processes" Basili & Rombach (1987)	Individual
Semantics error	"due to a misunderstanding or misuse of the semantic rules of a language" Basili & Rombach (1987)	Individual
Syntax error	"due to a misunderstanding or misuse of the syntactic rules of a language" Basili & Rombach (1987)	Individual
Environment errors	"due to a misunderstanding or misuse of the hardware or software environment of a given project." Basili & Rombach (1987)	Individual
Information Management errors	"due to a mishandling of certain procedures" Basili & Rombach (1987)	Tool|Individual
Phase-related	Causes relevant to each phase. This is not essentially a root cause category. It does not provide information about the actual causes but only the phase in which the fault was introduced.	Non-relevant
Project-related	"time pressure, management mistake, caused by other product. "	Tool|Individual
Review-Related	"no or incomplete review, not enough preparation, inadequate participation" Leszak, et al. (2002)	Individual

Distinct root cause category	Description	Actor
Human errors	Human mistakes due to cognitive limitations, insufficient knowledge and communication and etc.	Individual
Process errors	Bad process design, employees' non-compliance with defined processes, process non-compliance with the standards	Individual | process
Documentation errors	Mistakes in documenting and organizing the documents	Tool | Individual
Tool problems	Mistakes induced by tools directly such as Compiler induced defects	Tools
Task problems	Mistakes due to characteristics of the task such as ambiguity, difficulty, etc.	Individual
Input	Faults caused by wrong inputs	Tool | Individual

Tools can induce faults directly (Huang et al., 2012). These faults are not introduced due to an individual's misuse of the tool and its features; rather, they are caused by a malfunction or a problem in the tool itself. It is important to note that tools are, normally, developed outside of the company and their internal working mechanisms are not visible. This characteristic of tools places them out of the control of the company and means that tool-induced faults for the most part cannot be prevented.

Processes can induce faults directly as well. Existence of faulty processes or lack of a certain process can lead to faults slipping through to operation (Huang et al., 2012; Walia, & Carver, 2013). It is important to notice that process-induced faults are not due to wrong execution or non-compliance. It is the lack of a process and/or a defective process that can induce faults. An example of such a situation is when proper review and testing processes are not included at the 'transitions' between related life cycles (Van Moll, Jacobs, Freimut, & Trienekens, 2002). In this case, one can contend, if a fault is not detected and slips through to operation, it is induced directly by the process.

Processes hold certain similarities and differences with tools in terms of inducing faults. Processes are often adopted from a standard or are imposed by a regulatory body. In these cases, processes like tools are developed outside the company. However, unlike tools, internal mechanisms of processes are not invisible to the company. This distinction between the two means that individuals inside the company have control over the processes and are able to prevent process-induced faults. Defective processes can be identified and improved, and new ones can be introduced to fill their absence. These actions should take place by individuals inside the company and if they are not, then individuals are to be held accountable for them. Plus, it happens very rarely for processes to be fully adopted from standards. In fact, most standards and process models do not provide

detailed information of techniques to carry out an activity. It is the responsibility of the individuals within the company to adopt and tailor the processes to maximize performance and quality. Consequently, prevention of individuals' erratic behavior could prevent process-induced faults as well.

Individuals are the main actors who deliver faults. As is clear from TABLE 5, individuals appear in all root cause categories. This central role of individuals suggests that by preventing the individuals' erratic behaviors, faults can be prevented from slipping through to operation.

Focusing on individuals' erratic behaviors in order to prevent faults slipping through to operation is not a radically new idea and has been the subject of study in the literature before. Lanubile, Shull, and Basili (1998) presented a method called 'error abstraction' in which common patterns of individuals' erratic behaviors are abstracted from existing fault data. These common patterns, then, provide valuable input for better fault detection (Lanubile et al., 1998). The error abstraction method was later complemented by Walia and Carver (2013) who proposed taxonomy of individuals' erratic behaviors with respect to requirement faults. Even though, both Lanubile et al. (1998) and Walia and Carver (2013) have focused solely on requirement faults, and did not cover all development faults, there is promising evidence that common patterns exist in individuals' erratic behavior in other stages, such as coding, as well. Pan, Kim and Whitehead (2008), for example, demonstrated that developers consistently make mistakes when specific code situations occur. Moreover, Huang et al. (2015) showed that in the company they studied most of the severe and minor faults were caused due to human errors and noncompliance with processes.

Based on the findings that individuals, tools and processes contribute to faults slipping through to operation, the fault prevention model presented in FIGURE 2 was updated. FIGURE 4 depicts the updated model that includes the actors as well as contributing elements to faults slipping through to operation. FIGURE 4 presents a causal model similar to that of FIGURE 2. For example, an individual's action causes an ineffective fault fix which in turn causes a new fault to be introduced. This new fault can then slip through to operation.

In order to prevent individuals' erratic behavior, it is best to start with human errors in the existing root cause categories presented in TABLE 5. An investigation of human errors as presented by Huang et al. (2012) and Walia and Carver (2013) shows that individuals' erratic behavior occur partly due to cognitive constraints of the human mind and partly due to factors in the context. While cognitive constraints of the human mind are not malleable, introduction of appropriate practices can resolve the problems occurring due to factors in the context.

Consider constituents of human error category presented by Huang et al. (2012); among these, 'schema mismatching', 'working memory overload', 'evaluation error', and 'problem representation error' are caused due to the mechanics of the human mind. Such errors occur if the circumstances to which

they are vulnerable emerge; therefore, the focus should be on preventing those circumstances from emerging rather than changing the characteristics of the human mind! The same principle applies to the erratic behavior caused by "constraints on the cognitive abilities of requirement authors" (Walia and Carver 2012).

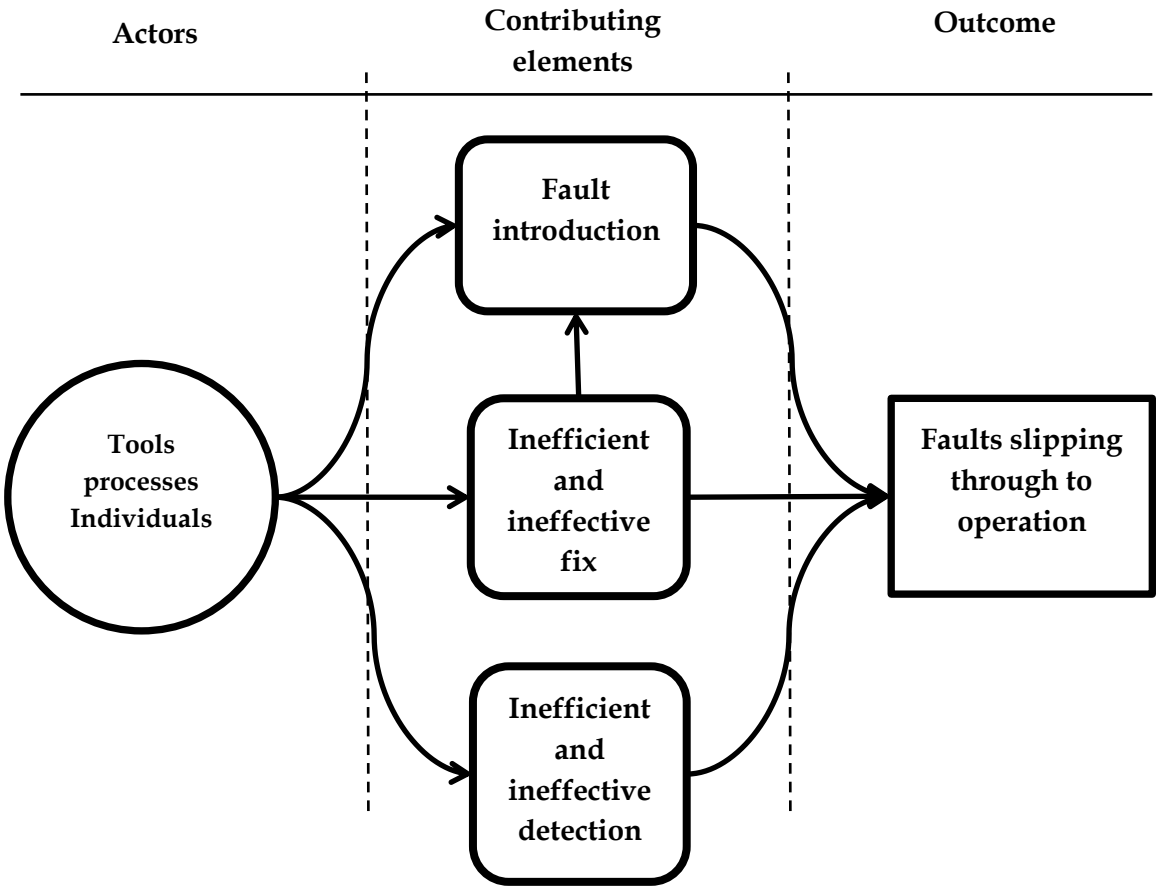

FIGURE 4 Actors in fault prevention model

The rest of the people errors, on the other hand, occur with respect to factors in the context. Shortage of knowledge, for example, as an error that was considered by all of the authors in TABLE 4, is relative to many factors in the context such as the complexity of the task, ambiguity of the task, domain of the project, application of the software system under development and programming language of choice, to name a few. Leszak et al. (2002) reported that mismatch between the skill-level needed in the project and the available skill of individuals can lead to systematic introduction of faults. A developer who is skillful in the domain of avionics might introduce faults when working in the domain of banking services. This developer's lack of domain knowledge can be catered for by introducing suitable practices such as adding extra reviews or training the developer. For example, Yu (1998)

reports a case in which "coding fault prevention guidelines" were introduced in a company to reduce faults slipping through to operation.

Communication problem was another category of people errors mentioned by the authors in TABLE 4. Communication problems occur with respect to communication mechanisms and the social context of the project. Managers can tailor their development processes to include practices that promote clear communication. Scrum's (Schwaber, & Beedle, 2002) daily stand up meetings, for example, can be introduced into a project for this purpose (Carpenter, & Dagnino, 2014). However, introduction of new practices should also be done in accordance with other contextual factors. Scrum's daily standup meetings might not be a suitable communication mechanism for an environment consisting virtual teams (Jacobs et al., 2005) or a project in which individuals have responsibilities in several projects (Sidky, & Arthur, 2007). In such cases, other measures should be considered such as acquisition of a secure online teleconference tool.

Inattention and tool misusing (Huang et al., 2012) occur with respect to the context of development, as well. For instance, one of the reasons the underlying reasons for inattention, according to Huang et al. (2012), is individuals' involvement in more than one task. Project pressure can also lead to inattention (Leszak et al. 2002). Clearly, project pressure and involvement in several projects are specific to a certain development context.

Procedure violation (Huang et al., 2012), on the other hand, except for the cases where it is done intentionally, is due to lack of procedure knowledge and inattention. Lack of procedure knowledge can occur in an environment where there is not enough training, or when documentation about procedures is not publicly available to everyone.

Drawing upon what was explained above, it can be postulated that human errors occur when contextual factors are not addressed with suitable practices. Individuals' erratic behaviors are for the most part raised due to mismatches between development context and existing development practices. Fenton and Neil (1999) claimed that while the mismatch between design effort and problem complexity leads to introduction of defects, the mismatch between design size and testing effort leads to ineffective detection of defects. Design effort and testing effort are development practices that are asked by Fenton and Neil (1999) to be matched to design size, and problem complexity that are factors of the context. Similarly, the mismatch between design effort and functionality was argued by Avižienis et al. (2004) as one of the prime causes of development failures.

If individuals' erratic behaviors indeed occur to mismatches between development practices and the context then it is reasonable to believe that mismatches can signal potential erratic behaviors. As a consequence, by identifying mismatches, one can essentially identify potential erratic behavior. It follows that such erratic behaviors can be prevented, simply, by tailoring the development method so that the development practices fit the context.

The interviewee's responses, indeed, showed that in the case company mismatches could potentially lead to erratic behaviors and eventually faults. For example, the interviewee explained a mismatch between time and resources and the practice of code reviewing. Such a mismatch results in abandoning code reviews in favor of catching deadlines and could lead to ineffective fault detection. Ineffective fault detection, in return, could lead to faults slipping through to operation.

> We have envisaged to use code review in projects and we finally don't have time to do it; or resources. (interview 1)

In another instance when asked about adopting agile practices the interviewee hinted that a mismatch between an organizational structure which allows a person to work in three projects and a communication channel of choice (Scrum's daily standup meetings in this case) could be problematic. Such a mismatch could lead to miscommunication and eventually a fault slipping through to operation.

> You cannot do daily [stand up] meetings […] when one person is working for three projects. (interview 1)

Moreover, possible erratic behavior regarding defect detection effectiveness was addressed when the interviewee pointed out a mismatch between ideal testing practices and the project size. (interview 1)

> If you are developing a software, it is difficult to define tests that will discover problems because the problems you can think about, you have already put in there. But, also in that case, we didn't have the size scale to have two separate parts of organization [testers and developers]. (interview 1)

Another example brought about was that of mismatch between tool support and coding standards. The interviewee expressed his concern that a lack of tool support would essentially lead to noncompliance with coding standards which is an erratic behavior.

> In general, we see that if you don't have an automatic way to perform this verification of your work, you end up not doing it; so even if we have tried to define some things in the past in practice they are not applied. (interview 1)

When asked about developers' compliance with defined procedures and guidelines, the interview noted that assignment of tasks should match the background and ways of working of developers. Otherwise, a mismatch between the two might result in noncompliance. (interview 1)

> It's a matter of also assigning, to each person working in the project, the activities that are more suited to the way of working, to the background. (interview 1)

Moreover, in another instance the interviewee implied that a mismatch between time pressure, and the audit practices could lead to minimal unit testing. Lack of sufficient unit testing could lead to faults slipping through to operation.

> I think it's more time pressure, because, well, let's say, for unit test which is probably the most useful, it requires a lot of maintenance and usually if you don't have someone behind that really sees that you invest some effort in doing unit testing, the result is that you end up testing for the, well, for having the system working more or less; you don't care about finding all the defects, but, you say when problem appears in the future, I will solve this specific problem; I won't invest effort in developing a test suite that will double my maintenance work. (interview 1)

The data from the interview confirms that in the case company mismatches between the context and development practices can signal potential erratic behavior of individuals which could lead to faults slipping through. Therefore, a proactive prevention solution is recommended based on identifying and resolving such mismatches by appropriate tailoring of the development method.

The idea of matching the software development to the needs of the context is well-established in the research community (Austin, & Devin, 2009; Hardgrave, Wilson, & Eastman, 1999; Iivari, 1989). According to Fitzgerald, Hartnett and Conboy (2006), method engineering and contingency theories dominate this research stream. While the contingency theory approach suggests selection of development to take place according to factors in the context from a portfolio of methods, the method engineering approach advocates method tailoring by adding and removing already verified and tested method 'fragments' (Fitzgerald et al., 2006).

In practice, there is indeed very little chance that a method is fully adopted and development methods are almost always subject to tailoring (Fitzgerald, Russo, & O'Kane, 2000). Tailoring allows adoption of practices according to contingencies in the context in order to maximize performance and quality (Austin, & Devin, 2009; Hardgrave, Wilson, & Eastman, 1999). Even standards are subject to tailoring. Fitzgerald et al. (2000) provided an example of a company that tailored the IEEE-std-1074 (1991) to the contingencies of the development context of each of their projects. Adoption of fault detection practices based on the contingencies in the context is not strange to tailoring either. Runeson, Andersson, Thelin, Andrews and Berling (2006), for example, recommended fault detection practices to be chosen based on Artifact, Types of defects, Actor, Technique, Purpose, Defect detection activity and Evaluation criteria factors. Sidky and Arthur (2007), on the other hand, proposed a three stage model to guide the selection of appropriate agile practices that fit the context of safety-critical projects.

The interviewee confirmed that tailoring of the standards does take place in the case company in the early stages of development based on criticality level of system under development.

> So usually there is request for performance and criticality analysis of the system. This means, in this space standards, four levels of criticality that depends on the consequences of the failure of the systems or parts of the system. So once you perform this analysis and decide what is your criticality level then it drives what practices you have to follow per standard. You tailor the standard. (interview 1)

In terms of quality, Fitzgerald et al. (2006) reported a case in which fault density was reduced by a factor of 7 by tailoring agile practices. Even though such achievement could be attributed to the use of agile methods, Fitzgerald et al. (2006) argued that the synergistic effect of tailoring could not be overlooked. Nevertheless, not much is known about practical ways of performing the tailoring (Fitzgerald et al., 2000). To respond to this challenge a number of studies have tried to identify the contextual factors that drive the tailoring of the development process (Bern, Pasi, Nikula, & Smolander, 2007; Clarke, & O'Connor, 2012). Even though the authors of such studies have done comprehensive work for identifying contextual factors, their findings are often too broad in scope and scale to be applicable in practice. Therefore, it would be beneficial if the scope is narrowed down to factors that affect faults slipping through to operation. To this end, in this research, taxonomy of contextual factors affecting faults slipping through to operation is developed. This taxonomy is instrumental to identifying and resolving mismatches between development context and practices which can in return drive the tailoring of the development process.

5.1.3 Objectives and solution

With RCA being the main instrument toward identification of root causes of faults, proactive RCA appears to be a legitimate goal to prevent faults from slipping through to operation. Based on the findings in the first phase of this research endeavor, the solution to such a proactive approach can capitalize on identifying mismatches between the context of development and practices. To this end, three objectives are defined as follows:

(1) developing a taxonomy of contextual factors that can affect faults slipping through to operation in the literature
(2) defining a method for proactive RCA (PRORCA)
(3) demonstrating the use of the PRORCA method

Completing the first objective assists the researcher and practitioners to get a strong foothold for identifying mismatches between development context and practice with respect to faults slipping through to operation. Without any doubt, the list of identified contextual factors is not a definitive list; nonetheless, it provides the research and practice community with a platform, upon which future activities can be launched. Developing taxonomy is well-justified since it is a

language for communication. Taxonomy of contextual factors that affect prevention of faults slipping through to operation helps differentiate between different factors and between factors and practices. The second objective is self-explanatory and upon its completion PRORCA as a proactive RCA method is developed. The first two objectives will be addressed in second phase of this research endeavor.

In design science research development of an artifact requires demonstration of its use (Peffers et al., 2007). Therefore, in phase three of this study the use of the PRORCA method will be demonstrated for two project of the case company. In the same phase the PRORCA method is evaluated.

5.2 Phase two

This phase of the research maps to the design and development stage in DSRM. The outcome of Phase two is taxonomy of contextual factors that can affect faults slipping through to operation and the PRORCA method. The taxonomy is an analyst's guiding light to find mismatches between context and practice which, itself, is the main tool at the analyst's disposal for proactive RCA. Development of the taxonomy was done using directed qualitative content analysis (Hsieh and Shannon 2005).

Content analysis is one of the semiotic modes of analysis used in qualitative research (Myers, 1997). Semiotics is a mode of analysis in which signs and symbols in language are scanned with the purpose of drawing conceptual categories from them (Myers, 1997). Such categories can be used for testing different aspects of a theory. Content analysis is used to understand or explain a phenomenon through a systematic process of coding and identifying patterned regularities in text (Hsieh, & Shannon, 2005; Myers, 1997). There exists three different approaches to content analysis, namely, conventional, directed, and summative (Hsieh, & Shannon, 2005). The main distinction between a directed content analysis and a conventional one is that in the directed approach, previous research findings or theory is used to initialize a set of predetermined categories (Hsieh, & Shannon, 2005). It is based on this set of predetermined categories that the researcher starts the coding process in order to understand or explain a phenomenon.

Since a set of predetermined categories initialize the analysis, a directed approach to content analysis can provide evidence for supporting or nonsupporting previous research findings or theory (Hsieh, & Shannon, 2005). Additionally, if new text is identified that cannot be coded according to the predetermined categories, new categories can be defined. "Newly identified categories either offer a contradictory view of the phenomenon or might further refine, extend, and enrich the theory"(Hsieh, & Shannon, 2005). Due to such

convincing practicalities of directed content analysis, it was adopted for developing taxonomy of contextual factors. Adoption of this technique resulted in a taxonomy that was developed by confirming and/or extending the predetermined code categories derived from the findings of previous research.

In order to define the initial set of code categories for developing the taxonomy of contextual factors, it was found necessary to determine the best way a development context can be understood. It was decided that such an understanding can be achieved by examining the context in terms of its constituent elements from different perspectives.

According to Sjøberg et al. (2008), in a typical software engineering situation "an actor applies technologies to perform certain activities on an (existing or planned) software system". From this statement, four key elements of a typical software engineering situation are understood to be actor, technology, activity and software system. On the other hand, people, processes and products (3Ps) as the key elements in software development have repeatedly been linked to quality of software (Allen, 2009; Norris, Rigby, & Stockman, 1994; Shah, 2014). The 3Ps are also the main categories of metrics used in fault prediction literature for predicting the number and occurrence of faults (Herrmann, 1998). The similarities between the four components of software engineering as identified by Sjøberg et al. (2008) and 3Ps are uncanny; the actors of Sjøberg et al. (2008) are in fact the people aspect of the 3Ps; the software system is the product; and the activities are the processes. Since it seemed to the researcher that the 3Ps were broader in scope compared to the four elements of Sjøberg et al. (2008), 3Ps were chosen as the constituent elements of software development.

Additionally, the development context can be described from different perspectives. One shall examine the context from all perspectives if they intend to get a complete view of it. It could be argued that (1) region, (2) organization, (3) project and (4) team perspectives can provide the best viewpoints to software development context. These perspectives, in fact, represent social groups of which an individual is a member of during software development. An individual could be a member of many social groups; however, these four groups are of significance because they relate to development. For instance, a developer can belong to a certain political party but there is little to no direct way that factors in the political party context can affect that developer's erratic behavior. Each of these social groups is a subgroup of the higher level group and can affect an individual's behaviors and preferences in a manner different from the other. Therefore, these four perspectives, region, organization, project and team, were considered sufficient to provide a window to the context of development.

To summarize, the four perspectives of context and the 3Ps of software development were considered to be two dimensions by which the context can be understood. While the 3Ps could guide us to understand the context in terms of its constituent's elements, the four perspectives of context could provide a

comprehensive view of it. These two dimensions were, therefore, selected as the predetermined code categories based on which directed content analysis (Hsieh, & Shannon, 2005) was conducted.

5.2.1 Taxonomy of contextual factors

The development of the taxonomy was carried out in four steps using directed content analysis (Hsieh, & Shannon, 2005). The initial coding categories necessary to perform directed content analysis were selected to be 4 perspectives of development context and 3Ps as key constituent elements of development. For the purpose of coding, a factor was defined as any phenomena, stimulant or circumstance that can be characterized as part of the context.

In order to identify the contextual factors eight distinct areas of research related to fault prevention were scanned. The academic articles reviewed in this phase were the same ones reviewed in phase one. However, a number of topic areas from phase one were excluded on the grounds of being irrelevant to fault prevention. Furthermore, a number of studies were removed. These studies were Zhang, & pham (2000) and Jacob et al (2006) and were left out because they introduced a set of factors which could influence the development of the taxonomy. TABLE 6 shows the topic areas and the number of papers in each topic area that was subject to directed content analysis (Hsieh and Shannon 2005).

TABLE 6 Topic areas investigated for developing taxonomy of contextual factors

Topic area	**number of articles reviewed**
fault detection	45
human factors	5
reliability modeling	6
Fault reporting and RCA	31
fault prediction (change analysis,)	24
defect analysis	7
fault reduction	17
software reliability engineering	6

The first step in conducting directed content analysis was highlighting all the phrases from each study that gave the impression of introducing or explaining a potential contextual factor. Hsieh and Shannon (2005), proposed that in cases where identifying all instances of a phenomenon is favorable, the coding begin after highlighting all the text that appears to represent that phenomenon. Highlighting the phrases in the text before starting to code increases trustworthiness of a

directed content analysis (Hsieh, & Shannon, 2005). A phrase was considered for highlighting if it belonged to any of the following groups of phrases.

- Phrases that specifically drew a causal relationship between fault introduction, detection and fix.
- Phrases that explained characteristics of the environment in which the study took place, the system that was the subject of study, and the people involved
- Phrases that explained a reason for error-proneness or fault proneness or drew a correlation between them
- Phrases that addressed improvement in failure rate, fault rate, severity, priority and reliability in general
- Phrases that directly or indirectly mentioned influential factors using terms like influence, affect, cause, result in and etc.
- Phrases that addressed lack of a practice, tool or phenomena and its consequences
- Phrases that addressed problems and difficulties in development
- Phrases that addressed efficiency and effectiveness of detection, analysis, and fix

The second step included scanning the highlighted phrases in the first step and coding potential contextual factors according to predetermined code categories. The predetermined code categories considered in this step were 3Ps. Coding according to four perspectives of context was performed later on. Upon identification of a factor, it was recorded in a master file along with the source in which it was found. Any factor that could not be categorized in accordance with the predetermined code categories was also recorded in the master file to be revised later. In total, a whopping 455 factors were found from 86 sources were recorded in the master file.

Phase three of content analysis was carried out to refine the 455 factors. Refining was initiated by removing duplicate factors and was later complemented by reexamining the uncategorized factors and redefining the predetermined codes. At this stage, it was revealed that 3Ps of context characteristics (people, process, and product) were inadequate to describe all the factors identified. After comparison and analysis of all factors, including the uncategorized ones, context characteristics were extended to human, artifact, environment, and activity.

Environment factors, as the name implies, refer to phenomena or stimulants in the surroundings of the people involved, the practices and deliverables. Factors related to high level strategies and supporting technologies are included as environment factors. Human factors are those relating to individual's characteristics, behaviors, duties and their interactions with other individuals. The activity factors characterize the context in terms of the practices carried out and the processes followed to develop a product. It is important to emphasize that these factors do not refer to technicality of activities and how they are done, rather the

existence and quality of activities that are known to affect the development practices and other contextual factors. Finally, artifact factors address characteristics of any deliverable produced during development. The artifact could be a simple document which is the outcome of requirement analysis, or design. It could be the source code or the whole software system in general. It is note-worthy that the context is dynamic. It changes synchronically as the development progresses. One cannot map the context according to the contextual factors statically and expect them to remain unchanged during the whole development. Therefore, it is necessary to treat the context as a living organism that needs care and nurturing. Mismatches between the context and development practices cannot all be figured out during the planning phase; rather they should be revisited as deemed necessary as the development progresses. All in all, after refining the taxonomy and duplicate removal, 144 factors remained.

The third step was coding the factors again; this time according to the four perspectives of context. The four perspectives of context adequately covered all the factors and as a result were left unchanged. Finally, in the fourth step, the identified factors were reexamined to find factors that were conceptually similar. This step was different than duplicate removal. The factors that are conceptually similar are not duplicate factors; rather they refer to different aspects of the same phenomena or stimulant that affects fault prevention. Consequently, conceptually-similar factors were merged by choosing an umbrella term to describe them.

At the end of the four steps of directed content analysis, 85 factors were identified and taxonomy of contextual factors was developed. Due to space limitations the complete taxonomy of contextual factors that can affect fault prevention is presented in appendix 1. An empty template of the taxonomy is presented as TABLE 7. On one dimension the perspectives of the context and on the other the constituent elements are visible. The taxonomy will aid an analyst in identifying mismatches between context and practices.

TABLE 7 template of the taxonomy of contextual factors

	Environment factors	**Artifact factors**	**Activity factors**	**Human factors**
Region				
Organization				
Project				
Team				

5.2.2 PRORCA method

RCA is one of the main instruments proposed in the literature to prevent faults from slipping through to operation. The PRORCA method is a solution to the difficulties of conducting RCA. It is developed to shift the predominantly reactive nature of RCA to proactive and to liberate it from reliance of fault data. The difficulties of conducting RCA are discussed in section 5.1.1 in more detail. The mechanism of the PRORCA method is adapted, to a large extent, from the ARCA method (Lehtinen et al., 2011) as it offers flexibility and freedom from fault data.

Founded on the finding that mismatches between context and practice can signal individual's erratic behavior, the PRORCA method comprises three steps: (1) context mapping (2) Erratic behavior mapping and, (3) Corrective action innovation. In the first step, the context of the development is mapped. This task can be completed using the taxonomy of contextual factors developed in section 5.2.1. In the second step, mismatches between the context and practices and in elements of the context itself, is identified and using causal maps (Bjørnson, Wang, & Arisholm, 2009) the relationship between mismatches and individuals' erratic behaviors will be mapped. In the last step, corrective actions will be introduced. These corrective actions will be derived from the mismatches mapped in the previous step.

Two roles are defined for carrying out PRORCA: the participant and the RCA facilitator. The distinction between the participant and the facilitator roles is in the logical design of the method and in reality the facilitator can take the role of the participant, as well, and vice versa. Such a design allows logical distribution of responsibilities between the participant(s) and the facilitator while allowing the responsibilities to be assigned to individuals flexibly with respect to available project resources and structure. The role of Facilitator is similar to that of Moderator in inspection (Aurum, Petersson, & Wohlin, 2002). The facilitator guides and controls the RCA. The participant, on the other hand, has the knowledge and know-how of context and practices of development. The facilitator enters the realm of the participant in order to make it possible for the participant to identify the mismatches, map the relationship between mismatches and erratic behaviors, and innovate corrective actions. It is essentially the participant(s) that fulfill(s) the goals of the RCA. The participant(s) can be any of the stakeholders in the development. Project managers, quality managers, analysts, designers, developers, testers, reviewers, team leaders could all take the role of participant. The decision of who actually becomes a participant depends on the resources available and is up to the facilitator to decide.

As discussed in section 5.1.2 individuals are actors whose erratic behaviors lead to fault introduction, ineffective, and inefficient detection and ineffective and inefficient fix. Identifying mismatches between the development context and practices can signal potential erratic behavior of individuals. Acquiring a good understanding of the context is necessary for identification of such mismatches. In step one of PRORCA, the goal is to map the context based on which mismatches

can be identified later. To this end, the RCA facilitator selects the participants and outlines meetings. The meetings can be in the form of qualitative interviews, focus group meetings, or any other form according to available resources. Plus, as proposed by Lehtinen et al. (2011), the data can be collected and handled either anonymously or publicly. The number of meetings is also a decision for the facilitator to take. If deemed sufficient for mapping the whole context, one meeting will wrap this step. Otherwise, further meetings are held. During the meeting(s), the facilitator provides the participant(s) with the taxonomy of contextual factors, presented in appendix 1, and records the participants' perceptions of the context. The recording could be done in an Excel file in which the value of each contextual factor is inputted in a key-value pair fashion. Any other method of recording is acceptable as long as the facilitator consider it sufficient. As soon as the facilitator perceives the context to be well understood, step one is complete. Mapping the context would help identifying the mismatches between the context and practices in the next step.

The prerequisite of step two is a good understanding of context and practices. So far, as a result of completing step one, the context has already been understood and the wise selection of participants has ensured a good knowledge of development practices. The second step is carried out with the purpose, firstly, to identify mismatches between the context and practices and, secondly, to map the relationship between mismatches and individuals' erratic behaviors. To this end, the RCA facilitator plans and holds meetings with the participants. Similar to step one, the form and the number of the meetings are flexible and are up to the facilitator to decide.

During the meetings, the participant(s), with the help of the facilitator, identify the mismatches. The mismatches can be written of post-it notes which later will be used for mapping. Next, the relationship between mismatches and erratic behaviors will be mapped using a causal map (Bjørnson et al., 2009) to potential erratic behaviors. The potential erratic behaviors, coupled with other mismatches, can lead to other erratic behaviors.

FIGURE 5 shows the template of a causal map. Causal maps are used in Lehtinen et al. (2011) and Bjørnson et al. (2009) to structure the cause-effect relationships. In PRORCA, a simple freehand variety of causal maps similar to the one used in Bjørnson et al. (2009) is used. The nodes, in this map, are either mismatches or individuals' erratic behaviors and each node appears just once in a causal map. The directed arrows, on the other hand, indicate cause-effect relationships. The arrows in PRORCA can indicate boolean operations if necessary. For example, in FIGURE 5, mismatch 1 and mismatch 2 together can cause erratic behavior 1. Or, alternatively, mismatch 3 can cause erratic behavior 1. Unlike, mapping the cause-effect relationships in Bjørnson et al. (2009), in which mapping is done knowing the problem, in PRORCA, the individuals' erratic behaviors are unknown beforehand. The information that is available for mapping in PRORCA is

the mismatches and the contributing elements to faults slipping through to operation. Therefore, in PRORCA the causal map is constructed bottom-up and proactively in the light of fault introduction, ineffective and inefficient detection and ineffective and inefficient fix.

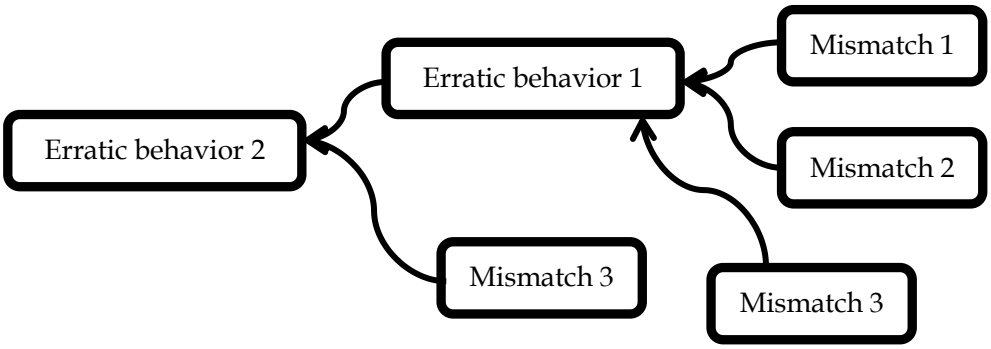

FIGURE 5 Causal map template

Both the facilitator and the participant(s) can draw upon their experience and knowledge to map the mismatches to erratic behaviors. The facilitator should promote discourse at this stage. The participant should convince the facilitator and other participants that an erratic behavior would occur due to a certain mismatch or combination of mismatches using reasonable arguments. If the participant manages to convince others of the possibility of an erratic behavior, the facilitator draws an arrow between the mismatch and that erratic behavior. Both the mismatches and erratic behaviors can be written on post-it notes and put on a board. This process will be iterated until the participants come into a conclusion that all mismatches are mapped to possible erratic behaviors.

The final step of PRORCA is innovation of corrective actions. Not much has been said in the RCA literature about developing corrective actions (Lehtinen et al., 2011). However, In PRORCA, innovating corrective actions is done in a straightforward manner by deriving corrective actions from the mismatches leading to individuals' erratic behaviors. In this step, the facilitator guides the participant(s) to innovate feasible corrective actions. Corrective actions should either prevent the emergence of circumstances that give rise to human cognitive constraints or resolve a mismatch. To this end, meetings are held by the facilitator, in accordance with the resources available. Yet again, the form and the number of the meetings are for the facilitator to decide. During the meetings the participant(s) prioritize the erratic behaviors. The prioritization of erratic behaviors drives the agenda of the meeting and the innovation of corrective actions.

It is note-worthy, that the three steps need not be conducted in separate meetings. In one single qualitative interview or focus group meeting all the steps can be completed. The facilitator should plan the PRORCA according to the available resources, and participants' schedules. This makes PRORCA a flexible

and lightweight method to be used in SMEs as well as large enterprises. As regards the appropriate time for carrying out PRORCA, the dynamic and constantly changing nature of the development context should be considered. In TABLE 3, it was shown that reactive RCA methods in the literature are recommended to be performed either after each stage or after each iteration (Bhandari et al., 1993, Jalote, & Agrawal, 2005). Kalinowski et al. (2008) added that RCA can be done in the wake of an unprecedented event as well. Following the reverse of these suggestions, PRORCA is recommended to be conducted before each major stage of development or iteration. Additionally, PRORCA can be conducted when signs of problems begin to surface in a project.

5.3 Phase three

In this phase firstly, the use of the PRORCA method was demonstrated in line with the purpose of the demonstration stage in DSRM (Peffers et al., 2007). To this end, the PRORCA method was applied in two ongoing projects of the case company in order to demonstrate its applicability in the field. The case company was the same company represented before. Later on, an evaluation of the PRORCA method was done at the end of this phase.

5.3.1 Demonstration

The PRORCA method is used in two small-scale projects of the case company. Small in this context refers to a project that one person can handle all the macro and micro development tasks (Boehm 1975).

Project 1 was a prototype project in the domain of avionics in which onboard software system prototypes were being developed to be used in future spacecrafts. Project two was also in the domain of avionics, however, in this project a system including both hardware and software was under development. This system is a replacement for a currently onboard system on the International Space Station (ISS), hence, a low level of tolerable risk and necessity for backward compatibility. In both cases, the software developer had close contact with the team leader and participated in meetings with their respective clients. At the time of conducting the PRORCA, project one was in late stages of development and mainly validation activities was taking place, while project two was still in the early stages. Therefore, corrective actions would still benefit both projects.

The PRORCA method comprise of three steps: (1) context mapping (2) Erratic behavior mapping and, (3) Corrective action innovation. For the purpose of mapping the development context, the RCA facilitator held two online interviews

with the main software developer in each project. In this step, the researcher took the role of the RCA facilitator and the main developer in charge of each project was the participant. The choice of online interview over focus group meetings or face-to-face meetings was made based on the availability of participants and the geographical distance between the researcher and the participants.

The interviewee for the first project was a software engineer with over 20 years of experience in defense and space industry. The interviewee had been working for the case company for four years at the time of the interview and was the fourth engineer assigned to this project in three years. The interviewee for the second project was a system engineer with more than six years of experience in telecommunication and space industries. He was responsible for design and development of the software and selection of the hardware in the second project which was running for over two years.

The interviews were semi-structured and in order to avoid interviewer bias, the questions were not directly addressing the context; rather they were general questions about how development was being carried out. In this step, as opposed to the guidelines of step one, the researcher did not share the taxonomy of contextual factors with the interviewees in any of the projects to avoid confirmation bias. However, in retrospect, the experience of the analysis of the data in later stages showed that sharing the taxonomy with the participants would have been constructive. The interviews were recorded and later transcribed.

In order to map the context, the text of the interviews was subject to directed content analysis (Hsieh, & Shannon, 2005). In directed content analysis, coding begins based on a set of predetermined code categories. Taxonomy of contextual factors, presented in appendix 1, was used for this purpose. At the end of this step, for each project an Excel file was created, holding the key-value pairs of contextual factors. Appendices 2 and 3 show the key value pairs for identified contextual factors for each project. It is important to note that due to time limitations, the interview was limited to the project and team perspectives of the taxonomy and did not cover region and organization perspectives.

In step two of PRORCA, erratic behaviors are mapped using directed graphs. Before the mapping starts, mismatches between practices and the context must be identified. Due to unavailability of further interviews with the company representatives in each project, the researcher took both RCA facilitator and participant roles to find the mismatches and map potential erratic behaviors. Needless to say that this was not an ideal situation but the time limitations of participants did not allow further meetings. In order to find the mismatches, the researcher used personal knowledge and experience to find the mismatches between key-value pairs available in the Excel files created at the end of step one. As soon as no further mismatches could be identified, mapping the erratic behaviors started. FIGURE 6 and FIGURE 7 demonstrate the map of erratic behaviors

and their relationship with mismatches and other erratic behaviors in the first and second project, respectively.

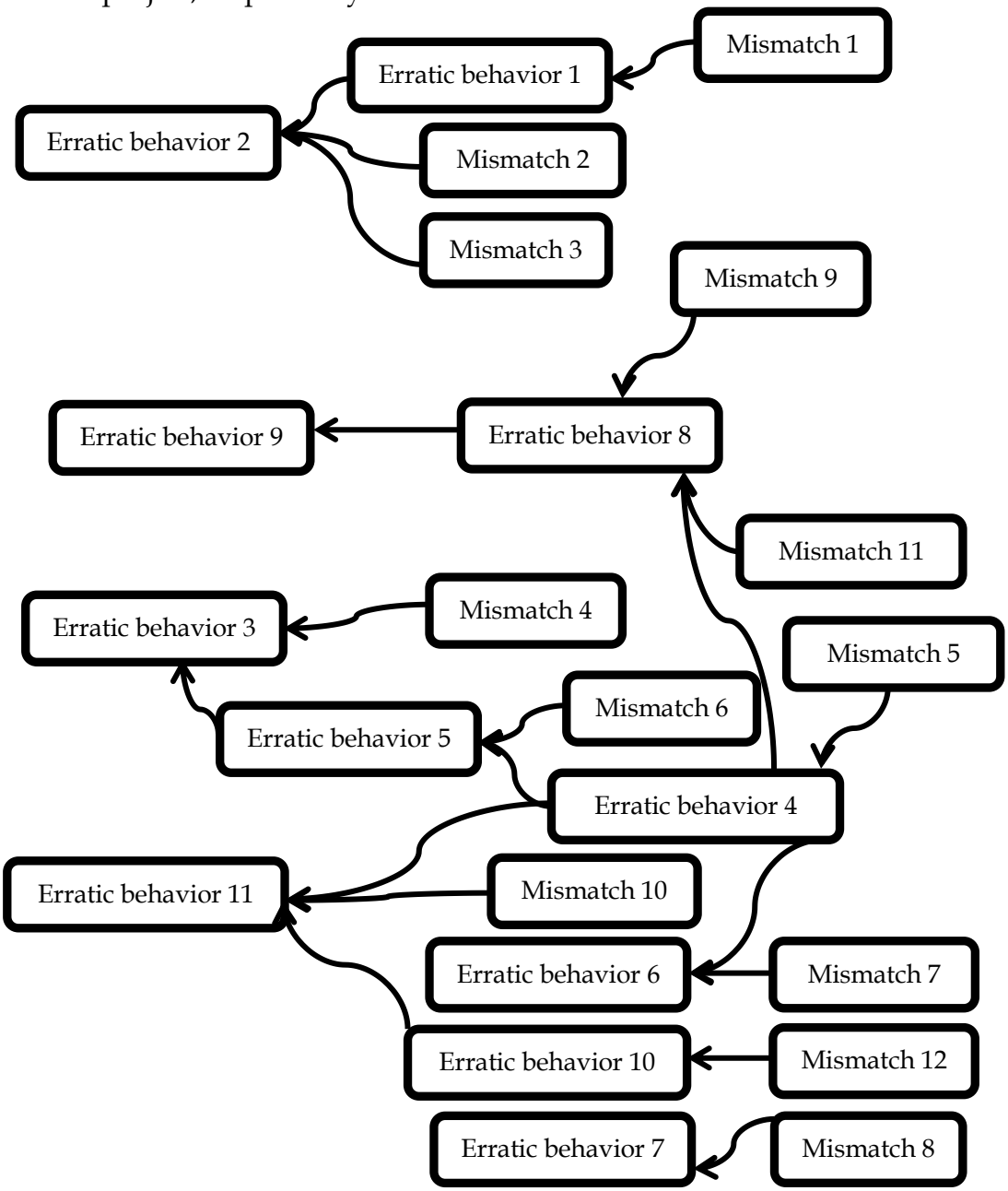

FIGURE 6 Project one causal map

The causal map of project 1 (FIGURE 6) shows three separate paths, each of which leading to different erratic behaviors. The first path is visible on the top of the figure and shows that mismatch 1 leads to erratic behavior 1. Furthermore, this path indicates that erratic behavior 1, mismatch 2 and mismatch 3 could result in

erratic behavior 2. The second main path in FIGURE 6, which is visible in the middle of the figure, depicts a complicated network of mismatches and erratic behaviors. Erratic behavior 4 stands out in this path, having a central role and leading to erratic behavior 5, 11 and 8. Lastly, the third path shows mismatch 8 leading to erratic behavior 7. This path can be viewed at the very bottom of the figure. Description of mismatches for the first project is provided in TABLE 8.

TABLE 8 Description of mismatches for the first project

Mismatch #	**Between**	**Description**
Mismatch 1	Project schedule and developer tasks	If the schedule is tight and the developer has a lot of tasks to complete, the level of commitment to defect data collection falls
Mismatch 2	level of commitment to defect data collection and the practice of prioritizing the defect fixes	It was stated by the interviewee that as the schedule becomes tighter, the level of commitment to defect data collection falls. Lack of commitment to defect data collection might primarily cause a problem when you consider that in the project defect fixes are prioritized. A defect that has not been reported might go unnoticed in planning and scheduling of fault fixes and subsequently slip through to the final product.
Mismatch 3	Level of commitment to defect data collection and staff changes	If faults are not collected properly, considering that the project has seen several staff changes before, there is a chance defects would go unnoticed
Mismatch 4	evolvability and frequency of changes in staff members	When members sacrifice commenting in expense of catching deadlines there is a threat that if a staff change occurs, the next person will have difficulty understanding what was supposed to go on, what was supposed to be developed and etc. Such misunderstanding of the works of previous developers can lead to introduction of faults.
Mismatch 5	Project schedule and documentation practices	Quality of documentation drops at the expense of catching deadlines
Mismatch 6	Reliance on documentation, quality of documentation and frequency of staff changes	If quality of documentation drops at the expense of catching deadlines then reliance on documentation can introduce problems. The interviewee however claimed that he relies less on documentation in the latest phases. This does not solve the problem however. If the documentation is not relied upon for development, then development becomes a matter of developer's experience and skills, considering the frequency of staff changes even if the current developer is highly skilled and experienced, the staff who are supposed to continue development in

Mismatch #	Between	Description
		future or maintain and update the product in future might inadvertently introduce faults.
Mismatch 7	Degree of trust in other members and documentation quality	High trust in what previous members have done coupled with documentation quality that drops at the expense of deadlines, might inhibit critical analysis of documentation and result in faults.
Mismatch 8	Availability of feedback with number of project members	Since there is only one person doing everything in this project, if that person does not receive constructive feedback, he is prone to not noticing his own mistakes. The interviewee admitted that this is not ideal. Even though the meetings with the customer can act as a feedback process, it might simply be too little too late.
Mismatch 9	Developer experience with DSDM and development method chosen	The interviewee stated that an agile development method called DSDM with a number of iterations were planned for the project in the beginning, he also admitted his lack of experience with this method. Had they actually stuck by their plans to develop using DSDM, such lack of experience with the chosen development method of the sole and main developer of the project could have led to ad-hoc development practices. However the interviewee mentioned that they went through one V-cycle at the end.
Mismatch 10	Intention to reuse in future and availability of definitions and guidelines	Even though the interviewee expressed his lack of information whether this prototype project would continue, he did express that they intend to reuse several components in future. If this is the case, then lack of high quality documentation and non-evolvability of source code could lead to introduction of faults. Plus definitions and guidelines would be necessary. As the interviewee mentioned they are not doing any extra effort.
Mismatch 11	Quality of documentation, reliance on documentation and timespan between updates:	As the interviewee mentioned some inconsistencies in the documents goes unnoticed until they are reported by the customer, in such a case the inconsistencies are fixed in next stages, however, this lag coupled with non-reliance on documentation toward the final stages by the developer might come at a high price of developing using ad-hoc processes. Some things might be forgotten or go unnoticed.
Mismatch 12	Commenting practices and project schedule	When members sacrifice commenting in expense of catching deadlines

Descriptions of erratic behaviors for the first project, presented in FIGURE 6, are provided in TABLE 9. The last column of the table includes the cause of each erratic behavior. As mentioned before each erratic behavior can be caused by different causes. For example, erratic behavior 3 can be caused either by mismatch 4 or erratic behavior 5. Alternatively, existence of mismatches and erratic behaviors

might have a synergistic effect leading to other erratic behaviors. Erratic behavior 5, for instance, is cause by the existence of both erratic behavior 4 and mismatch 6.

TABLE 9 Description of erratic behaviors for the first project

Erratic behavior #	Description	Cause
Erratic behavior 1	Noncompliance with defect reporting procedures	Mismatch 1
Erratic behavior 2	Defects go unnoticed in planning and scheduling of defect fixes	Erratic behavior 1 and mismatch 2 and mismatch 3
Erratic behavior 3	Lack of sufficient knowledge about the current state of development	Mismatch 4 or Erratic behavior 5
Erratic behavior 4	Noncompliance with documentation procedures	Mismatch 5
Erratic behavior 5	Non-reliance on documentation for development	Erratic behavior 4 and mismatch 6
Erratic behavior 6	Lack of critical analysis of documents	Mismatch 7 and Erratic behavior 4
Erratic behavior 7	Not noticing self-mistakes	Mismatch 8
Erratic behavior 8	Non-compliance with the development method and defined procedures	Mismatch 9 or Erratic behavior 4 and mismatch 11
Erratic behavior 9	Delayed delivery	Erratic behavior 8
Erratic behavior 10	Non-evolvability of the source code	Mismatch 12
Erratic behavior 11	Possible reuse of components without sufficient knowledge	Erratic behavior 10 and Erratic behavior 4 and mismatch 10

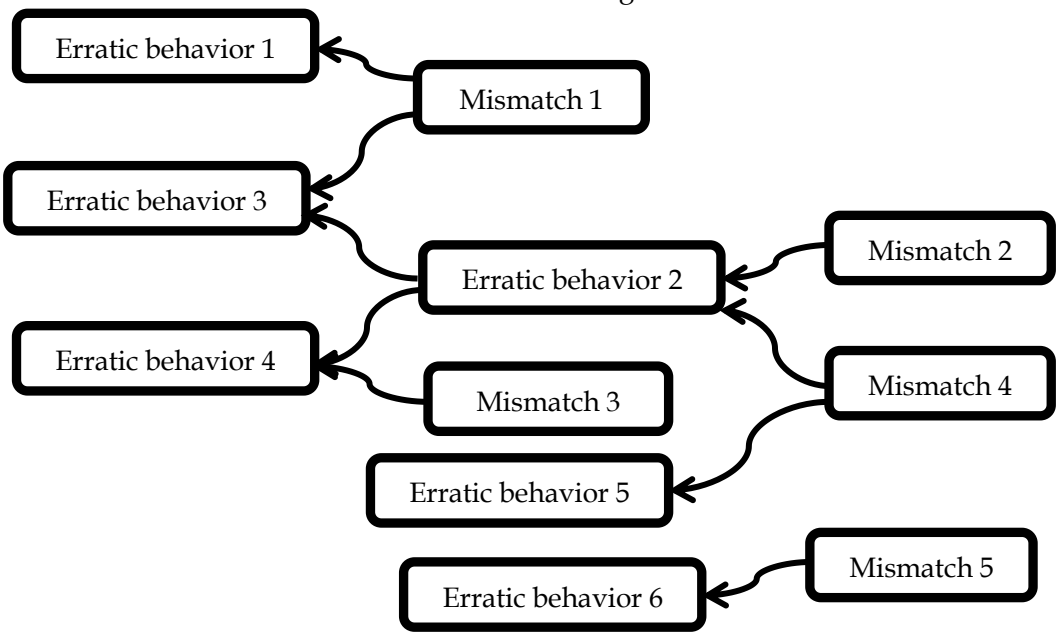

FIGURE 7 Project two causal map

The causal map of project 2 (FIGURE 7) shows two separate paths. The first patch depicted on the top shows the interconnections between mismatches 1, 2, 3, 4 and erratic behaviors 1, 2, 3, 4, 5. The second path, visible on the bottom of the figure, shows the potential cause-effect relationship between mismatch 5 and erratic behavior 6. Description of mismatches for the second project is provided in TABLE 10.

TABLE 10 Description of mismatches for the second project

Mismatch #	**Between**	**Description**
Mismatch 1	Necessity of backward compatibility and concurrency of development	While part of the system needs to hold backward compatibility with scripts developed by experiment container developers, since these developers are working in parallel to the team, no such script is provided to the development team. This inconsistency could result in faults in the form of unsupported previous behavior.
Mismatch 2	Selection of defect detection practices and reliance on customer feedback	Late reviews held with the customer and reliance on such meetings for feedback results in long time-span between updates to documents and late delivery
Mismatch 3	Reliance on documentation and time-span between updates to documents	Considering that the developer relies heavily on documentation, long time-span between updates to documents might lead to faults being introduced
Mismatch 4	Availability and quality of documentation and tool support	The interviewee wished for better tool support for documentation. In case the tool is difficult to use and handle, considering that high quality and availability of documentation is necessary for this project and considering that the developer relies heavily on documentation, this inconsistency might lead to improper handling or update of the document and eventually a fault.
Mismatch 5	Availability of feedback with number of project members	No one inside the company is reviewing the works of the developer, this means that the point of departure is meetings and reviews with the customer, however such meetings are too little, too late.

Descriptions of erratic behaviors for the second project, presented in FIGURE 7, are provided in TABLE 11. The last column of the table includes the cause of each erratic behavior.

TABLE 11 Description of erratic behaviors for the second project

Erratic behavior #	Description	Cause
Erratic behavior 1	Development without regards to requirements	Mismatch 1
Erratic behavior 2	Late update of documents	Mismatch 2 or mismatch 4
Erratic behavior 3	Delayed delivery	Erratic behavior 2 or Mismatch 1
Erratic behavior 4	Development based on incorrect information	Erratic behavior 2 and mismatch 3
Erratic behavior 5	Non-compliance with documentation procedure	Mismatch 4
Erratic behavior 6	Not noticing self-mistakes	Mismatch 5

The goal in the last step of PRORCA is innovation of corrective actions. The corrective actions can be derived from the mismatches in order to prevent erratic behaviors. Yet again for completing this step, the researcher was both the participant and the RCA facilitator. In this step corrective actions could be prioritized so that a sudden change of routines does not distress the development. Since, deriving corrective actions without further knowledge of possibilities in the project is hardly possible, and because no further interviews were available, only few examples of possible corrective actions were innovated in this step.

Looking at the causal map of the first project developed in the previous step indicates that 'erratic behavior 4' is of central importance. Since only 'mismatch 5' is considered as the root of this erratic behavior, the top priority in project one should be addressing that mismatch. To this end, the recommended corrective actions could be (1) changing the documentation practices or (2) increasing the number of developers in order to decrease the workload of the current developer.

In the second project, 'erratic behavior 2' has a central role. Since 'mismatch 2' or 'mismatch 4' could lead to this erratic behavior, solutions should address both of these mismatches. Corrective action for 'mismatch 2' could be adding extra review sessions with internal reviewers. On the other hand, 'Mismatch 4' could be resolved by introducing new documentation tools or recruiting new members into the project to care for and handle the documentation.

In this step the researcher, demonstrated the straight-forward manner in which corrective actions can be innovated for 'erratic behavior 4' in the first project and 'erratic behavior 2' in the second project as examples. Corrective actions for other erratic behaviors can be generated in the same fashion.

5.3.2 Evaluation

Evaluation of the PRORCA method is done in terms of the difficulties of carrying out RCA outlined in section 5.1.1. First and foremost, it should help the analyst

prevent faults proactively. Secondly, it should be resource-friendly. Furthermore, it should benefit SMEs as well as larger companies. And, lastly, the degree of its vulnerability to fault reporting mechanisms should be low.

PRORCA with its reliance on mismatches between the development context and practices is not only theoretically proactive, but, proved to be proactive in practice too. In section 5.3.1 potential erratic behaviors that could lead to fault introduction, ineffective and inefficient detection and ineffective and inefficient fix were identified in two ongoing projects using PRORCA and corrective measures were recommended. This feature of PRORCA is in stark distinction to the existing RCA methods in the literature that are all reactive in nature. This is, by no means, diminishing the value of other RCA methods; rather it shows that PRORCA can complement reactive RCA methods by adding a forward feed to fault prevention activities.

Resource-wise, PRORCA benefits from the same advantages attributed to ARCA method (lehtinen et al., 2011) and more. It can be as lightweight or as heavyweight as necessary. It was demonstrated, in section 5.3.1, that even one interview with a knowledgeable participant can surface erratic behaviors. The total amount of time spent on each of the interviews was 90 minutes only. Mismatch identification and erratic behavior mapping for both of the projects took less than 4 man hours. Plus, non-reliance of PRORCA on fault data means there is no need for heavy startup costs. It is, also, liberating for the staff because there would be no need for the time-consuming task of reporting faults in accordance with a specific classification schema (Chillarege, et al., 1992). It is evident that PRORCA is not vulnerable to the problems inherent in fault reporting mechanisms because it does not mandate collection of fault data.

Resource-friendliness, flexibility and non-reliance on fault data are characteristics that make PRORCA not only applicable in large companies but also SMEs. In fact the case company in this research is an SME and the size of each project was small in terms of the number of project staff. But still, it was demonstrated that projects and companies of this size can benefit from applying PRORCA. All in all, PRORCA is not subject to difficulties of applying reactive, fault-data-oriented RCA methods.

Even with due consideration of all of its merits, PRORCA is vulnerable to being left out of fault prevention mechanisms in companies due to its invisible gains. Since PRORCA is a proactive method and does not address an existing problem, it can easily be considered as a luxurious practice by management. The situation is exacerbated by the lack of empirical results approving its effectiveness in preventing faults from slipping through to operation. For these reasons, longitudinal studies investigating the benefits of applying RCA and its effect on fault rate and fault severity is necessary. This task, however, is left to future research for now.

The taxonomy of contextual factors developed in this research endeavor is instrumental to conducting PRORCA. However, it is by no means exhaustive or finalized. As an example, one might think that language as a contextual factor should be included in the region perspective. Even though, it is arguable that language is covered in the taxonomy as the communication factor represented in the project perspective, the staff of a company might think otherwise. In such a case the taxonomy provides enough flexibility for the language factor to be added to the region perspective. In fact, based on the knowledge and experience of the staff of the context, the taxonomy can be customized in a way to represent the context in the best possible way.

6 Discussion

This research is an extension to the works of Lanubile et al. (1998), Jacobs et al. (2007), Lehtinen et al. (2011), and Clarke and O'Connor (2012). Furthermore, the model of fault prevention presented in this study holds familiarity to fault prediction literature.

Similar to the 'Error abstraction' method proposed by Lanubile et al. (1998), in this study the main underlying theme is identification of common individual errors. However, while the error abstraction method relies on abstracting common errors from a set of already existing faults, in this research the goal is identification of individuals' erratic behaviors with due consideration to the mismatches between the context of development and practices. It is arguable that this difference between the two studies marks a difference in a reactive approach in Lanubile et al. (1998) and a proactive approach in this study. Another point of departure between the two is the scope of application. Lanubile et al. (1998) focused solely on requirement faults; however, fault prevention should be extended to all stages of development. Identification of individuals' erratic behaviors in this study is done proactively for all development stages.

As regards mapping the context, Clarke and O'Connor (2012) developed a reference framework of situational factors that can be used as a tool for defining software development processes or to deliver improvements. Jacobs et al. (2007), on the other hand, developed lists of factors that can positively or negatively affect fault introduction and fault detection. The taxonomy of contextual factors developed in this research effort is similar to the situational factors reference framework of Clarke and O'Connor (2012) in providing a tool for mapping the context of development. However, in doing so, the taxonomy of contextual factors, presented in this research, limits factors to those that can affect fault prevention in terms of fault introduction, inefficient and ineffective detection and inefficient and ineffective fix, hence, is similar to the work of Jacobs et al. (2007). Narrowing down the scope of the taxonomy improves its utilization for the purpose of finding the

mismatches between the development context and practices. The reference framework of Clarke and O'Connor (2012) has 8 factor classifications, 44 factors, and 172 sub-factors. The large scale of this framework compared to 85 factors and two dimensions presented in the taxonomy of contextual factors render it inapplicable for the purposes of this study. Even though 85 contextual factors might still be too many to handle in practice, since the taxonomy is presented in two dimensions, practitioners can focus on the dimensions which they find most important.

Even though, lists of influential factors identified by Jacobs et al. (2007) is suitable in scope and scale, the authors' focus on influential factors limits the applicability of their findings for identification of mismatches in the context. Admittedly, though, the contextual factors identified in this research turned out to hold many similarities to the influential factors of Jacobs et al. (2007). Consequently, practitioners unwilling to adopt the taxonomy of contextual factors for mismatch identification might benefit from investigating the influential factors of Jacobs et al. (2007) as a replacement.

The PRORCA method is an extension of the ARCA method (Lehtinen et al. 2011). While the ARCA method (Lehtinen et al., 2011) was designed to be lightweight, resource-friendly and applicable in SMEs, it still assumed the existence of problems and emphasized on reactive identification of problems. The PRORCA method, while benefiting from the merits of the ARCA method (Lehtinen et al., 2011) is designed to be proactive. It does not assume the existence of problems. Instead, PRORCA can be used on a quest to identify and resolve erratic behaviors that could potentially contribute to faults slipping through to operation. Another, point of departure between the two methods is that in the ARCA method (Lehtinen et al., 2011), problems are under investigation, but in PRORCA, the goal is fault prevention. The problems that the ARCA method (Lehtinen et al., 2011) seeks to resolve include but are not limited to faults. However, there is no reason why the PRORCA method cannot focus on broader problems than faults. This would be a matter of future studies though.

The PRORCA method focuses on individuals' erratic behavior, however, as was shown in section 5.1.2, tools and processes can also lead to faults slipping through to operation. The contribution of tools and processes to faults slipping through to operation shall not be overlooked. Companies should maintain active communication lines with tool vendors and report possible misbehaviors of tools. Such communication is done with the goal of tool improvement. Similarly, process contribution to faults slipping through to operation shall be recorded and reported to standardization and regulatory bodies. Models and standards are not free from faults by design, they are someone's ideal of a development process (Grady, 1996). The company's active participation in a consortium and reporting their experience can, therefore, contribute to fixing the defective processes and filling the absence of others, in their source.

The emphasis of the PRORCA method on future faults brings similarities with fault forecasting (Avižienis et al., 2004; Lyu, 1996) to mind. However, the focus in fault forecasting (or fault prediction) literature is on making measurements of the current state of the software or development process in order to estimate the future number or presence of faults using statistical analysis (Catal, & Diri, 2009; Fenton, & Neil, 1999; Hall et al., 2012). Such analysis can be used to predict the number of faults in a module, file or a piece of software in terms of fault density and fault rate (Fenton, & Neil, 1999; Fenton, & Ohlsson, 2000). Alternatively, the analysis can be done to distinguish fault-prone and non-fault-prone modules (Munson, & Khoshgoftaar, 1992) and files (Hammil, & Goseva-Popstojanova, 2009) or to rank pieces of software according to their fault-proneness (Zhou, & Leung, 2006). The measurements required for such analysis can be done according to different sets of metrics. The most common metrics found are extracted from fault reports (Fenton, & Neil, 1999). The idea is to extract information like defect rate from fault reports and perform mathematical extrapolation. In recent years, efforts have been taken to make fault predictions based on the number and type of changes made in the source code (Graves, Karr, Marron, & Siy, 2000; Kidwell, Hayes, & Nikora, 2014; Kim et al., 2006; Ostrand, Weyuker, & Bell, 2005). In cases where fault data is not available researchers have suggested product, process and people metrics (Herrmann, 1998). Product metrics are those that are extractable from an artifact of the software product. Product metrics include size (Shen, Yu, Thebaut, & Paulsen, 1985) and complexity metrics (Mccabe 1976) and object-oriented design metrics (Nugroho, & Chaudron, 2014) to name a few. People metrics deal with the development team and other stakeholders involved (Herrmann, 1998). Process metrics try to make predictions based on the effectiveness and efficiency of development processes (Leszak, et al. 2002). Clearly, the PRORCA method does not fit this profile. In PRORCA the potential future faults are predicted by identification of individuals' erratic behaviors. The PRORCA is not performed in order to estimate the number of faults with respect to size (fault density), or to distinguish between fault-prone and non-fault-prone modules. The ultimate goal in PRORCA is to resolve the mismatches between development context and practices to prevent future faults. There is no denying, though, that fault prediction studies can contribute to PRORCA, however, studying the possibilities raised from such contribution is left to future research for now.

It could not be overstated that rather than being a replacement for the existing RCA methods in the literature, the PRORCA method is a complement to them. While the benefits of reactive RCA has already been well-recognized, proactive RCA can provide a feed forward for companies to look into the future and see what potential problems might lay ahead.

7 Limitations

This study has a number of limitations. Firstly, material extraction for the mapping study was subject to reliability concerns. The forward and backward search technique deployed in this study is vulnerable to the comprehensiveness of the initial set of articles chosen and tends to favor backward search. To address these concerns, an expert interview was conducted in phase one. Even though the input of the interviewee did effectively direct material extraction for the better, still, the task was completed subjectively by the researcher.

Furthermore, the reliability of the taxonomy of contextual factors developed in this study was subject to vulnerabilities. The directed content analysis conducted for the purpose of developing the taxonomy was done by one researcher, solely. This is not ideal and may introduce biases in coding. In such a situation a contextual factor might be missed or wrongly included. It is arguable, however, that the large number of factors coded alleviates the problem of missing a factor by increasing the chance of covering it in analysis of other studies. On the other hand, there is a high chance for the wrongly included factors to have been dropped during the later steps of the development of the taxonomy.

Other major limitations were faced in the demonstration phase. The two projects for which PRORCA was conducted were not sufficient in scale and scope to surface all the potential difficulties of the method. Both projects were small in size. Project one was a prototype project for which the level of tolerable risk was fairly high. Even project two, in which, the level of tolerable risk was low, the focus was on hardware rather than software.

Other limitations in demonstration stage included few numbers of and short time available for interviews. Due to time limitations, the interviews were limited to the project and team perspectives of the taxonomy and did not cover region and organization perspectives. Additionally, in steps two and three of the PRORCA, the researcher took both RCA facilitator and participant roles because no further interviews were possible. Even though this situation showed the flexibility of

PRORCA, it evidently limited the possibility of finding erratic behaviors and innovating corrective actions.

Lastly, the interviewees might have been biased to present a better picture of the project than reality. The interviewees were the responsible individuals for their respective projects and it is understandable if they viewed the interviews as evaluation of their work. Plus, the projects were both held with representatives of one company and software reliability is a sensitive topic in software development, specifically in the domain of avionics and embedded software. Therefore, it could be possible that some information was withheld from the researcher.

8 Conclusion

In this research, the task of developing a proactive RCA method was undertook in response to a call for further research into fault prevention by Alho and Mattila (2011). The PRORCA method as the main outcome of this endeavor is lightweight, flexible and proactive and relies on finding the mismatches between the development context and the practices to prevent faults from slipping through to operation. Proactive prevention of faults slipping through to operation is done with the ultimate purpose of developing fault-free software systems. Even though, development of a system that is completely free from faults is far from reality, prevention of faults slipping through to operation can still make considerable contributions to development of highly reliable systems.

The PRORCA method was developed in accordance with the teachings of DSRM (Peffers et al., 2007) and comprises three steps (1) context mapping (2) Erratic behavior mapping and, (3) Corrective action innovation. In step one, the context of development is mapped using taxonomy of contextual factors. The taxonomy of contextual factors affecting faults slipping through to operation was developed in this research as well as the PRORCA method. The taxonomy consists of two dimensions – four perspectives of context and four constituent elements of development. Four perspectives of context are region, organization, project and team and the four constituent elements of context are environment, activities, artifacts and humans. In PRORCA's second step, potential erratic behaviors of individuals that can lead to faults slipping through to operation are mapped, in a bottom up approach, using directed graphs. The erratic behaviors are mapped based on mismatches between the development taxonomy and practices identified at the beginning of this step. Step three includes deriving corrective actions in accordance to mismatches that lead to erratic behaviors.

The use of the PRORCA method was demonstrated in this research in two small software development projects in the domain of avionics and embedded systems. Admittedly, these two projects were not sufficient in scale and scope to

surface all the potential difficulties and problems of the PRORCA method, however, they did prove the flexibility and resource-friendliness of the method.

REFERENCES

Adams, E. N. (1984). Optimizing preventive service of software products. *IBM Journal of Research and Development*, *28*(1), 2-14.

Alho, P., & Mattila, J. (2011). Dependable control systems design and evaluation. In Conference on Systems Engineering Research (CSER) 2011.

Allen, M. (2009). From substandard to successful software. *CrossTalk*, *22*(4), 29-32.

Aurum, A., Petersson, H., & Wohlin, C. (2002). State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, *12*(3), 133-154.

Austin R., & Devin, L. (2009). Weighing the Benefits and Costs of Flexibility in MakingSoftware: Toward a Contingency Theory of the Determinants of Development Process Design. Information Systems Research, 20(3), 462-477.

Avižienis, A., Laprie, J. C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, *1*(1), 11-33.

Babu, P. A., Kumar, C. S., & Murali, N. (2012). A hybrid approach to quantify software reliability in nuclear safety systems. Annals of Nuclear Energy, 50, 133-140.

Basili, V. R., & Rombach, H. D. (1987, March). Tailoring the software process to project goals and environments. In *Proceedings of the 9th international conference on Software Engineering* (pp. 345-357). IEEE Computer Society Press.

Beizer, B. (1990) Software Testing Techniques. International Thomson Computer Press

Bern, A., Pasi, S. J. A., Nikula, U., & Smolander, K. (2007, June). Contextual Factors Affecting the Software Development Process–An Initial View. In *2nd AIS SIGSAND European Symposium on Systems Analysis and Design, Gdansk, Poland, June* (Vol. 5).

Bhandari, I., Halliday, M., Tarver, E., Brown, D., Chaar, J., & Chillarege, R. (1993). A case study of software process improvement during development. *Software Engineering, IEEE Transactions on*, *19*(12), 1157-1170.

Binder, R. (2000). Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Professional.

Bishop, P. (2013). Does Software Have to Be Ultra Reliable in Safety Critical Systems?. In *Computer Safety, Reliability, and Security* (pp. 118-129). Springer Berlin Heidelberg.

Bjørnson, F. O., Wang, A. I., & Arisholm, E. (2009). Improving the effectiveness of root cause analysis in post mortem analysis: A controlled experiment. *Information and Software Technology*, *51*(1), 150-161.

Boehm, B. W., Mcclean, R. K., & Urfrig, D. E. (1975). Some experience with automated aids to the design of large-scale reliable software. *Software Engineering, IEEE Transactions on*, (1), 125-133.

Børretzen, J. A. *Software Fault Reporting Processes in Business-Critical Systems* (Doctoral dissertation, Norwegian University for Science and Technology).

Børretzen, J. A., & Dyre-Hansen, J. (2007). Investigating the software fault profile of industrial projects to determine process improvement areas: an empirical study. In *Software Process Improvement* (pp. 212-223). Springer Berlin Heidelberg.

Børretzen, J. A., Stålhane, T., Lauritsen, T., & Myhrer, P. T. (2004, November). Safety activities during early software project phases. In *Proceedings, Norwegian Informatics Conference*.

Bridge, N., & Miller, C. (1998). Orthogonal defect classification using defect data to improve software development. Software Quality, 3(1), 1-8.

Butler, R. W., & Finelli, G. B. (1993). The infeasibility of quantifying the reliability of life-critical real-time software. *Software Engineering, IEEE Transactions on*, *19*(1), 3-12.

Canfora, G., & Cerulo, L. (2005, September). Impact analysis by mining software and change request repositories. In *Software Metrics, 2005. 11th IEEE International Symposium* (pp. 9-pp). IEEE.

Card, D. N. (1998). Learning from our mistakes with defect causal analysis. *Software, IEEE*, *15*(1), 56-63.

Card, D. N. (2005). Defect Analysis: Basic Techniques for Management and Learning. Advances in Computers, 65, 259-295.

Carman, D. W., Dolinsky, A. A., Lyu, M. R., & Yu, J. S. (1995, October). Software reliability engineering study of a large-scale telecommunications software system. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on* (pp. 350-359). IEEE.

Carpenter, S. E., & Dagnino, A. (2014, September). Is Agile too Fragile for Space-Based Systems Engineering?. In Space Mission Challenges for Information Technology (SMC-IT), 2014 IEEE International Conference on (pp. 38-45). IEEE.

Carrozza, G., Pietrantuono, R., & Russo, S. (2015). Defect analysis in mission-critical software systems: a detailed investigation. *Journal of Software: Evolution and Process*, *27*(1), 22-49.

Catal, C., & Diri, B. (2009). A systematic review of software fault prediction studies. *Expert systems with applications*, *36*(4), 7346-7354.

Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., & Wong, M. Y. (1992). Orthogonal defect classification-a concept for in-process measurements. Software Engineering, IEEE Transactions on,18(11), 943-956.

Christenson, D. A., & Huang, S. T. (1996). Estimating the fault content of software using the fix-on-fix model. *Bell Labs Technical Journal*, *1*(1), 130-137.

Clarke, P., & O'Connor, R. V. (2012). The situational factors that affect the software development process: Towards a comprehensive reference framework. *Information and Software Technology*, *54*(5), 433-447.

CMMI Product Team. (2010). CMMI for Development, Version 1.3 (CMU/SEI-2010-TR-033). Retrieved February 13, 2016, from the Software Engineering Institute, Carnegie Mellon University website: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9661

Diaz, M., & Sligo, J. (1997). How software process improvement helped Motorola. *IEEE software*, *14*(5), 75.

Dunn, W. R. (2004). Software safety and reliability.

ECSS-Q-HB-80-03A. (2012). Space product assurance–Software dependability and safety ECSS-Q-HB-80-03A:2012. The European Space Agency Requirements & Standards Division

El Emam, K., & Wieczorek, I. (1998, November). The repeatability of code defect classifications. In Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on (pp. 322-333). IEEE.

Favarò, F. M., Jackson, D. W., Saleh, J. H., & Mavris, D. N. (2013). Software contributions to aircraft adverse events: Case studies and analyses of recurrent accident patterns and failure mechanisms. *Reliability Engineering & System Safety*, *113*, 131-142.

Fenton, N. E., & Neil, M. (1999). A critique of software defect prediction models. Software Engineering, IEEE Transactions on, 25(5), 675-689.

Fenton, N. E., & Ohlsson, N. (2000). Quantitative analysis of faults and failures in a complex software system. Software Engineering, IEEE Transactions on,26(8), 797-814.

Fitzgerald, B., & O'Kane, T. (1999). A longitudinal study of software process improvement. *IEEE software*, *16*(3), 37.

Fitzgerald, B., Hartnett, G., & Conboy, K. (2006). Customising agile methods to software practices at Intel Shannon. *European Journal of Information Systems*, *15*(2), 200-213.

Fitzgerald, B., Russo, N., & O'Kane, T. (2000). An empirical study of system development method tailoring in practice. *ECIS 2000 Proceedings*, 4.

Freimut, B., Denger, C., & Ketterer, M. (2005, September). An industrial case study of implementing and validating defect classification for process improvement and quality management. In Software Metrics, 2005. 11th IEEE International Symposium (pp. 10-pp). IEEE.

Goel, A. L. (1985). Software reliability models: Assumptions, limitations, and applicability. *Software Engineering, IEEE Transactions on*, (12), 1411-1423.

Grady, R. B. (1996). Software failure analysis for high-return process improvement decisions. Hewlett Packard Journal, 47, 15-24.

Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. (2000). Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, *26*(7), 653-661.

Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, *38*(6), 1276-1304.

Hamill, M., & Goseva-Popstojanova, K. (2009). Common trends in software fault and failure data. Software Engineering, IEEE Transactions on, 35(4), 484-496.

Hanmer, R. S., McBride, D. T., & Mendiratta, V. B. (2007). Comparing reliability and security: Concepts, requirements, and techniques. Bell Labs Technical Journal, 12(3), 65-78.

Hannay, J. E., Sjøberg, D. I., & Dybå, T. (2007). A systematic review of theory use in software engineering experiments. *Software Engineering, IEEE Transactions on*, *33*(2), 87-107.

Hardgrave, B. C., Wilson, R. L., & Eastman, K. (1999). Toward a contingency model for selecting an information system prototyping strategy. *Journal of Management Information Systems*, *16*(2), 113-136.

Harter, D. E., Kemerer, C. F., & Slaughter, S. A. (2012). Does software process improvement reduce the severity of defects? A longitudinal field study. *Software Engineering, IEEE Transactions on*, *38*(4), 810-827.

Hayes, J. H., Raphael, I., Holbrook, E. A., & Pruett, D. M. (2006, August). A case history of International Space Station requirement faults. In Engineering of Complex Computer Systems, 2006. ICECCS 2006. 11th IEEE International Conference on (pp. 10-pp). IEEE.

Herrmann, D. S. (1998, January). Sample implementation of the Littlewood holistic model for assessing software quality, safety and reliability. In *Reliability and Maintainability Symposium, 1998. Proceedings., Annual* (pp. 138-148). IEEE.

Herrmann, D. S., & Peercy, D. E. (1999, January). Software reliability cases: the bridge between hardware, software and system safety and reliability. In *Reliability and Maintainability Symposium, 1999. Proceedings. Annual* (pp. 396-402). IEEE.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75-105.

Hong, G. Y., Xie, M., & Shanmugan, P. (1999). A statistical method for controlling software defect detection process. *Computers & industrial engineering*, *37*(1), 137-140.

Hsieh, H. F., & Shannon, S. E. (2005). Three approaches to qualitative content analysis. *Qualitative health research*, *15*(9), 1277-1288.

Huang, F., Liu, B., & Huang, B. (2012). A taxonomy system to identify human error causes for software defects. In *Proceedings 18th Issat International Conference on Reliability & Quality in Design, Boston, USA* (pp. 44-49).

Huang, F., Liu, B., Wang, S., & Li, Q. (2015). The impact of software process consistency on residual defects. *Journal of Software: Evolution and Process*, *27*(9), 625-646.

Huber, J. T. (2000). A comparison of IBM's orthogonal defect classification to Hewlett Packard's defect origins, types, and modes. In Proceedings of International Conference on Applications of Software Measurement. San Jose, CA (pp. 1-17).

IEC 61508. (2010). Functional safety of electrical/electronic/programmable electronic safety-related systems. IEC 61508 ed. 2.0, International Electrotechnical Commission

IEEE Std 1044 (2009). IEEE Standard Classification for Software Anomalies.

IEEE Std 729 (1983). IEEE Standard Glossary of Software Engineering Terminology

IEEE Std-1074 (1991). IEEE Standard for Developing Software Life Cycle Processes.

Iivari, J. (1989). A methodology for IS development as organizational change: A pragmatic contingency approach. *Information Systems Development for Human Progress in Organisations, Amsterdam: North-Holland*, 197-217.

ISO, I. (2005). IEC 25000 Software and system engineering–Software product Quality Requirements and Evaluation (SQuaRE)–Guide to SQuaRE. *International Organization for Standarization*.

ISO, I. (2008). IEC 12207 Systems and software engineering-software life cycle processes. *International Organization for Standardization: Geneva*.

ISO, I. (2010). *IEEE, Systems and Software Engineering--Vocabulary*. ISO/IEC/IEEE 24765: 2010 (E)) Piscataway, NJ: IEEE computer society.

Jacobs, J., Van Moll, J., Krause, P., Kusters, R., Trienekens, J., & Brombacher, A. (2005). Exploring defect causes in products developed by virtual teams. *Information and Software Technology*, *47*(6), 399-410.

Jacobs, J., Van Moll, J., Kusters, R., Trienekens, J., & Brombacher, A. (2007). Identification of factors that influence defect injection and detection in development of software intensive products. *Information and Software Technology*, *49*(7), 774-789.

Jalali, S., & Wohlin, C. (2012, September). Systematic literature studies: database searches vs. backward snowballing. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement* (pp. 29-38). ACM.

Jalote, P., & Agrawal, N. (2005, December). Using defect analysis feedback for improving quality and productivity in iterative software development. InInformation and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on (pp. 703-713). IEEE.

Kalinowski, M., Travassos, G. H., & Card, D. N. (2008, September). Towards a defect prevention based process improvement approach. In Software

Engineering and Advanced Applications, 2008. SEAA'08. 34th Euromicro Conference (pp. 199-206). IEEE.

Kidwell, B., Hayes, J. H., & Nikora, A. P. (2014, October). Toward Extended Change Types for Analyzing Software Faults. In *Quality Software (QSIC), 2014 14th International Conference on* (pp. 202-211). IEEE.

Kim, S., Zimmermann, T., Pan, K., & Whitehead Jr, E. J. (2006, September). Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on* (pp. 81-90). IEEE.

Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., & Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, *28*(8), 721-734.

Kitchenham, B., & Charters, S., (2007) "Guidelines for Performing Systematic Literature Reviews in Software Engineering (Version 2.3)," Technical Report EBSE-2007-01, Keele Univ., EBSE.

Krishnan, M. S., & Kellner, M. I. (1999). Measuring process consistency: Implications for reducing software defects. *Software Engineering, IEEE Transactions on*, *25*(6), 800-815.

Lanubile, F., Shull, F., & Basili, V. R. (1998, November). Experimenting with error abstraction in requirements documents. In *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International* (pp. 114-121). IEEE.

Lehtinen, T. O., Mäntylä, M. V., & Vanhanen, J. (2011). Development and evaluation of a lightweight root cause analysis method (ARCA method)–field studies at four software companies. Information and Software Technology,53(10), 1045-1061.

Leszak, M., Perry, D. E., & Stoll, D. (2002). Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software*, *61*(3), 173-187.

Leveson, N. G. (2004). Role of software in spacecraft accidents. *Journal of spacecraft and Rockets*, *41*(4), 564-575.

Leveson, N. G., & Turner, C. S. (1993). An investigation of the Therac-25 accidents. Computer, 26(7), 18-41.

Levy, Y., & Ellis, T. J. (2006). A systems approach to conduct an effective literature review in support of information systems research. *Informing Science: International Journal of an Emerging Transdiscipline*, *9*(1), 181-212.

Li, B., Sun, X., Leung, H., & Zhang, S. (2013). A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, *23*(8), 613-646.

Li, N., Li, Z., & Sun, X. (2010, December). Classification of software defect detected by black-box testing: An empirical study. In *Software Engineering (WCSE), 2010 Second World Congress on* (Vol. 2, pp. 234-240). IEEE.

Littlewood, B., & Strigini, L. (1993). Validation of ultrahigh dependability for software-based systems. *Communications of the ACM (CACM)*, *36*(11), 69-80.

Lutz, R. R., & Mikulski, I. C. (2004). Empirical analysis of safety-critical anomalies during operations. *Software Engineering, IEEE Transactions on*, *30*(3), 172-180.

Lyu, M. R. (1996). *Handbook of software reliability engineering* (Vol. 222). CA: IEEE computer society press.

Lyu, M. R. (2007, May). Software reliability engineering: A roadmap. In 2007 Future of Software Engineering (pp. 153-170). IEEE Computer Society.

Mays, R. G., Jones, C. L., Holloway, G. J., & Studinski, D. P. (1990). Experiences with defect prevention. *IBM Systems Journal*, *29*(1), 4-32.

McCabe, T. J. (1976). A complexity measure. Software Engineering, IEEE Transactions on, (4), 308-320.

Mellegard, N., Staron, M., & Torner, F. (2012, November). A light-weight defect classification scheme for embedded automotive software and its initial evaluation. In Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on (pp. 261-270). IEEE.

Mohagheghi, P., Conradi, R., & Børretzen, J. A. (2006, May). Revisiting the problem of using problem reports for quality assessment. In *Proceedings of the 2006 international workshop on Software quality* (pp. 45-50). ACM.

Munson, J. C., & Khoshgoftaar, T. M. (1992). The detection of fault-prone programs. Software Engineering, IEEE Transactions on, 18(5), 423-433.

Musa, J. D. (1996). Software reliability-engineered testing. Computer, 29(11), 61-68.

Myers, M. D. (1997). Qualitative research in information systems. *Management Information Systems Quarterly*, *21*(2), 241-242.

Norris, M., Rigby, P., & Stockman, S. (1994). Life after ISO 9001: British Telecom's approach to software quality. *Communications Magazine, IEEE*, *32*(10), 58-63.

Nugroho, A., & Chaudron, M. R. (2014). The impact of UML modeling on defect density and defect resolution time in a proprietary system. *Empirical Software Engineering*, *19*(4), 926-954.

Okoli, C., & Schabram, K. (2010). A guide to conducting a systematic literature review of information systems research. *Sprouts Work. Pap. Inf. Syst*, *10*, 26.

Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, *31*(4), 340-355.

Pan, K., Kim, S., & Whitehead Jr, E. J. (2009). Toward an understanding of bug fix patterns. Empirical Software Engineering, 14(3), 286-315.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems*, *24*(3), 45-77.

Radjenović, D., Heričko, M., Torkar, R., & Živkovič, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, *55*(8), 1397-1418.

Raninen, A., Toroi, T., Vainio, H., & Ahonen, J. J. (2012). Defect data analysis as input for software process improvement. In Product-Focused Software Process Improvement (pp. 3-16). Springer Berlin Heidelberg.

Runeson, P., Andersson, C., Thelin, T., Andrews, A., & Berling, T. (2006). What do we know about defect detection methods?. *IEEE software*, *23*(3), 82.

Schwaber, K., & Beedle, M. (2002). gilè Software Development with Scrum.

Shah, S. M. A. (2014). *EMPIRICAL CHARACTERIZATION OF SOFTWARE QUALITY* (Doctoral dissertation, Politecnico di Torino).

Shen, V. Y., Yu, T. J., Thebaut, S. M., & Paulsen, L. R. (1985). Identifying error-prone software—an empirical study. Software Engineering, IEEE Transactions on, (4), 317-324.

Shenvi, A. A. (2009, February). Defect prevention with orthogonal defect classification. In *Proceedings of the 2nd India software engineering conference* (pp. 83-88). ACM.

Sidky, A., & Arthur, J. (2007, March). Determining the applicability of agile practices to mission and life-critical systems. In Software Engineering Workshop, 2007. SEW 2007. 31st IEEE (pp. 3-12). IEEE.

Sjøberg, D. I., Dybå, T., Anda, B. C., & Hannay, J. E. (2008). Building theories in software engineering. In *Guide to advanced empirical software engineering* (pp. 312-336). Springer London.

Vallespir, D., Grazioli, F., & Herbert, J. (2009). A framework to evaluate defect taxonomies. In XV Congreso Argentino de Ciencias de la Computación.

Van Moll, J. H., Jacobs, J. C., Freimut, B., & Trienekens, J. J. M. (2002, October). The importance of life cycle modeling to defect detection and prevention. In *Software Technology and Engineering Practice, 2002. STEP 2002. Proceedings. 10th International Workshop on* (pp. 144-155). IEEE.

van Moll, J., Jacobs, J., Kusters, R., & Trienekens, J. (2004). Defect detection oriented lifecycle modeling in complex product development. *Information and Software Technology*, *46*(10), 665-675.

Voas, J. M., & Miller, K. W. (1995). Software testability: The new verification.IEEE software, 12(3), 17-28.

Walia, G. S., & Carver, J. C. (2013). Using error abstraction and classification to improve requirement quality: conclusions from a family of four empirical studies. *Empirical Software Engineering*, *18*(4), 625-658.

Walls, J. G., Widmeyer, G. R., & El Sawy, O. A. (1992). Building an information system design theory for vigilant EIS. *Information systems research*, *3*(1), 36-59.

Webster, J., & Watson, R. T. (2002). Analyzing the past to prepare for the future: Writing a. *MIS quarterly*, *26*(2), 13-23.

Whittaker, J. A. (2000). What is software testing? And why is it so hard?.Software, IEEE, 17(1), 70-79.

Yu, W. D. (1998). A software fault prevention approach in coding and root cause analysis. *Bell Labs Technical Journal*, *3*(2), 3-21.

Zahedi, F. (1987). Reliability of information systems based on the critical success factors-formulation. *Mis Quarterly*, 187-203.

Zelkowitz, M. V., & Rus, I. (2004). Defect evolution in a product line environment. Journal of Systems and Software, 70(1), 143-154.

Zhou, Y., & Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *Software Engineering, IEEE Transactions on*, *32*(10), 771-789.

APPENDIX 1 TAXONOMY OF CONTEXTUAL FACTORS

Region	
Environment factors **Human factors**	Time zone culture (collective behavior and behavioral norms, differences in mental models)
Organization	
Environment factors	org size; org domain; organization strategy: improvement of the quality, lower cost org structure (resulting in communication delays) involvement of external organizations
Activity factors	Maturity of processes (change in processes, data tracking and management practices) quality of intra-project communication Quality of analysis on inter-related projects (impact analysis, RCA, etc.) Quality of fault reporting process Existence of Reactive/Proactive processes
Human factors	trust in other projects Involvement of staff on several projects reactive/proactive thinking organizational culture (continual improvement, reactive thinking)
Project	
Environment factors	project size project structure (complexity of organization's projects: multi-project development, single project development, or etc.) standards in place; budget and schedule Involvement of different stakeholders Degree of customer involvement level of tolerable risk office ergonomics, tool support education and training
Artifact factors	programming language used and its features criticality of subsystems scope of system's possible behaviors Testability quality of defect reports

	operational usage design problem history (persistency and stability), fault proneness of modules expected lifetime of system product size product complexity (clarity of interactions between subsystems) application domain backward compatibility Availability and Quality of documentation (requirements, design, test cases, etc.) volatility of requirements source code evolvability defect classification scheme used for fault reporting defect profile modeling paradigm
Activity factors	information flow between requirements and tests; interaction of developers with testing staff division of responsibilities between teams quality of communication within project assignment and handling of priorities selection process of defect detection practices (test approach and strategy) Independent defect detection availability of definitions and guidelines degree of compliance with guidelines and standards availability of feedback size of developed increments; alignment of testing and requirement analysis availability and usage of automated tests synchronicity of communication quality of defect and test case tracing Time span between updates to documents Concurrency of activities (code modification, coding and testing, etc.) coordination of testing activities development strategy (multiple release, open source, reuse, distributed and concurrent development, virtual development) defect fixing strategy (fixing low severity defects later and attend to high severity for now);
Human factors	Frequency of change in staff members level of commitment to defect data collection staff knowledge, skill and experience degree of attention to detail and priority of procedures fear of data misuse; Commitment toward high-quality development Degree of trust in other staff defined responsibilities conflicting schedules of experts

	availability of full time testing staff the number of people working on the project
Team	
Environment factors	team size Virtual/Co-located teams
Human factors	Frequency of change in team members team variation of skills and experience commitment to teamwork tendency of teams toward production blocking and evaluation apprehension; reviewers' collusion group synergy

APPENDIX 2 THE CONTEXT OF PROJECT ONE

Project		
Environment factors	project size	Small
	project structure	no other related projects
	standards in place	tailored ECSS E-40 standards
	Involvement of different stakeholders	minor support from other engineers
	Degree of customer involvement	Regular meetings
	level of tolerable risk	prototype project so high
	office ergonomics	cubicles and ergonomics campaign
	tool support	Redmine
	education and training	Non
Artifact factors	Testability	Not considered
	operational usage	Prototype
	expected lifetime of system	NA
	product size	Not large
	product complexity (clarity of interactions between subsystems)	Low, subsystems and interactions are known
	application domain	Avionics
	backward compatibility	NA
	Availability and Quality of documentation	high at the beginning but low at the end, might be problems because members have left
	volatility of requirements	low
	source code evolvability	A matter of schedule, no reviews for this matter. But the customer has specific requirement for percentage of comments
	defect classification scheme used for fault reporting	Provided by Redmine tool
	modeling paradigm	UML
Activity factors	interaction of developers with testing staff	NA
	division of responsibilities between teams	1 Person responsible for all tasks
	quality of communication within project	good but still miscommunication is reported

	assignment and handling of priorities	At the beginning of the project for bug fixes but a chance they would be ignored later
	selection process of defect detection practices (test approach and strategy)	No defined process
	Independent defect detection	No
	availability of definitions and guidelines	Available in certain cases like coding rules but no official procedure to review compliance.
	availability of feedback	No, customer reviews
	synchronicity of communication	synchronic with project leader, not synchronic with the customer
	Time span between updates to documents	either immediately or next release
	coordination of testing activities	NA
Human factors	Frequency of change in staff members	Frequent
	level of commitment to defect data collection	time pressure can stop collection
	staff knowledge, skill and experience	High
	Degree of trust in other staff	High
	defined responsibilities	
	availability of full time testing staff	No
	the number of people working on the project	2
Project		
Environment factors	Virtual/Co-located teams	Co-located

APPENDIX 3 THE CONTEXT OF PROJECT TWO

Project		
Environment factors	project size	small
	standards in place	tailored ECSS E-40 standard
	budget and schedule	schedule is very tight
	Involvement of different stakeholders	Sub-contractors and customers
	Degree of customer involvement	High
	level of tolerable risk	Low
	office ergonomics,	all members in one office
	tool support	Redmine, Doors, Doxygen
	education and training	Non
Artifact factors	programming language used and its features	C, C++
	scope of system's possible behaviors	predictable by using a state machine
	Testability	No, no time for analysis
	operational usage	known by operational scenarios
	expected lifetime of system	10 years
	product size	Large
	product complexity (clarity of interactions between subsystems)	A lot of interfaces and challenges of open source libraries
	application domain	onboard flight system
	backward compatibility	Yes
	Availability and Quality of documentation	everything in word docs
	volatility of requirements	volatile for new parts of the system
	source code evolvability	coding rules are defined, Doxygen documentation style
	defect classification scheme used for fault reporting	That of Redmine tool
	modeling paradigm	Not a model-driven development
Activity factors	information flow between requirements and tests;	Doors is used for manageability
	division of responsibilities between teams	no division. One person is responsible for all
	quality of communication within project	high, daily standup meeting, meetings within the project; with subcontractor and customer and phone calls and emails

	selection process of defect detection practices	mainly oriented around customer requirements (reviews) but tests are designed in-house
	Independent defect detection	Yes, another team will test the system in the end but not at this stage
	availability of feedback	Non, reliance on customer feedback
	synchronicity of communication	synchronous with other members of the project, synchronous with subcontractors, bi-weekly meeting with customer,
	Concurrency of activities	Yes, concurrent with customer and sub-contractors, experiment container developers (customer) are working in parallel
	development strategy	heavy use of open source software and libraries
Human factors	Frequency of change in staff members	No staff change
	Degree of trust in other staff	High
	availability of full time testing staff	No
	the number of people working on the project	3
Team		
Environment factors	team size	Small
	Virtual/Co-located teams	Co-located

APPENDIX 4 INTERVIEW QUESTIONS FOR INTERVIEW ONE

I. Background

1. Could you please introduce yourself and let us know about your background and role at [company name]?
2. Could you please briefly introduce [company name]?

II. General information

1. Could you please explain the development method currently being practiced at [company name]?
2. Are there any contractors involved in the development? For example in coding, testing, etc.
3. How are sub-projects and development of sub-components dealt with (Contractors, separate teams in a serial manner, teams working in parallel, distributed development)? How does this affect the development method?
4. How frequently are components reused at [company name] or are they at all? Do reused components go through a defect detection process too?
5. What standards are complied with?
6. Could you please explain the verification and validation practices at [company name]?
7. What mechanisms are in place at [company name] to help developer[1] teams prevent, detect and remove faults?
8. What are the general practices at [company name] to make sure developers comply with practices and policies?
9. Are there practices in [company name] that promote and encourage developers to enhance their personal disciplines? (education, training, feedback on frequent mistakes)

III. Detailed questions

Agile methods

1. Does [company name] have any experience with or considered using agile methods and/or practices for development? For example, pair programming, Test-Driven Development, scrum sprints, daily stand-up meetings, etc.
2. If yes, how are such practices chosen and adopted?

Customer reliability requirements

[1] The term developer refers to anyone involved in development of a system including analysts, designer, coders, testers and etc.

3. How does [COMPANY NAME] determine customer reliability requirements? For example, the customer asks for a certain reliability level, certification standards determine the necessary reliability, or by contacting the customer and extracting the requirements from discussions.
4. How is reliability measured at [COMPANY NAME]?
5. Is criticality analysis of functions and components performed at [COMPANY NAME]? Is there a difference between critical components and non-critical ones in terms of development and reliability practices?
6. Are the most frequently used functions of a system under development identified?

Fault data and changes

7. How do you deal with changes during development at [COMPANY NAME][2] ? Do you have mechanisms like a Change Control Board (CCB), use agile processes, or there is a customer proxy involved in the project?
8. Are defects, failures and changes traceable? What are the mechanisms used? What tools are used? How do you ensure that the traces are kept up to date (Is there a certain role that is responsible for keeping them up to date or each developer must make sure he/she submits the changes to a repository)?
9. How fault data is collected at [COMPANY NAME] or is it at all? What tools are used?
10. Could you please explain briefly what sort of information is collected for defects? Do developers fill in different forms at different stages of development?
11. How often does the structure of fault reports change or does it at all?
12. Who is responsible for defect detection (testers, inspectors, all project stakeholders)?
13. Who can report defects or failures (coders, designers, testers, sales persons, or anyone in the company)? Who has access to tools for fault reporting?
14. How is fault reporting enforced? What happens if a developer does not fill in fault reports or does not provide the necessary information?
15. In general how do you see developers' perception of fault reporting (as an overhead or important part of work)?
16. Does the quality of defect data filled in by developers allow analysis of data or is it ambiguous, too coarse-grained, etc.?
17. What kind of analysis is performed on fault/change data at [COMPANY NAME]?
18. Do you look for root causes of problems (frequent, severe, etc.) at [COMPANY NAME]? How? Do you perform Root Cause Analysis?
19. Do you deliver process improvements to prevent faults? How?

Defect detection

20. How is testing performed (in-house testing department or testing team, independent contractors)?

[2] A change can be a requirement change, an enhancement request, refactoring, fault removal, etc.

21. Do you use theorem proving and/or model checking techniques for verification? Do you use any tools helping with that?
22. Is inspection performed at [COMPANY NAME] in order to detect defects?
23. Could you please explain a typical inspection meeting?
24. Who do the inspection teams consist of?
25. At what points during development an inspection meeting is held (after each milestone, each sprint, iteration, etc.)?
26. Is analysis of defect data used to guide defect detection?
27. Are test cases traceable? What tools are used? Who is responsible?
28. How is the testing strategy determined?
29. Are static code analysis tools used?

Developer communication

30. What are the communication mechanisms between developers, used at [COMPANY NAME] (Official meeting, unofficial meetings, Scrum standup meetings)? What are the tools that enable such communication?
31. In particular what mechanisms exist in [COMPANY NAME] to allow testers and other developers (coders, designer, analysts, etc.) communicate? For example, how do the developers let testers know of changes? Are testers involved in early planning stages?
32. Is testability considered during requirements specification, design, and coding?

Other practices

33. What are the commenting practices at [COMPANY NAME]? What if a developer does not comply with commenting policies or best practices? How do make sure comments are kept up to date?
34. Are there any coding standards defined for coders? How are they enforced? What if someone does not comply?

APPENDIX 5 INTERVIEW QUESTIONS FOR INTERVIEW TWO AND THREE

I. Background

1. Could you please introduce yourself and let us know about your background and role at [company name] and the project you are involved with?
2. Could you please briefly introduce [company name] and the project?
3. What is the application domain of the system under development?
    - 3.1 How many people (approx.) are working on the project?
    - 3.2 How complex is the system under development? (scope of system's possible behaviors large or small, interactions between system's sub-systems, etc.)
    - 3.3 How large is the system under development?
    - 3.4 What is the expected lifetime of the system?
    - 3.5 Are there any external parties involved in the development? For example in coding, testing, etc.
    - 3.6 Is independent defect detection performed in the project?
    - 3.7 Contractors?
    - 3.8 Is it a multiple release project or just one release at the end of the project?
    - 3.9 Who/what is the user of the system being developed?
    - 3.10 Is the operational usage known to developers including the frequency of usage?
    - 3.11 Do you need to take backward compatibility in mind?
    - 3.12 How does the customer get involved in the project? During, before and after.

II. General information

4. Could you please explain the development method currently being practiced at the project?
    - 4.1 Do you use agile methods and/or practices for development? For example, pair programming, Test-Driven Development, scrum sprints, daily stand-up meetings, etc. If yes, how are such practices chosen and adopted?
    - 4.2 How frequently are components reused at [company name] or are they at all?
    - 4.3 Do reused components go through a defect detection process too?
5. How are the teams managed (assigned responsibilities) in the project in which you are involved (division of responsibilities between teams, etc.)?
    - 5.1 Is there a virtual development environment? Do you have virtual teams?
6. What precautions are taken to reduce the number of faults introduced?

6.1 Do you care for testability during development (all stages)?
6.2 Do you look for root causes of failures and faults? Do you perform Root Cause Analysis? Is there a defined feedback process (for example to let developers know what type of mistakes they have made and etc.)?

7. What are the defect detection practices used in the project? (testing strategies, testing techniques, type of reviews, people involved, , automatic scripts, etc.) How are they chosen?
    7.1 How are test activities coordinated?
    7.2 Are lower and upper bounds for defects detected during reviews?
8. What are the mechanisms to ensure high quality of documentation?
    8.1 How much do you rely on documentation in the project?
    8.2 How long does it take for a document (requirement, design, etc.) to be updated if there is any change?
    8.3 How committed are project members to document?
    8.4 What are the defect reporting mechanisms?
    8.5 How good are the defect reports in terms of quality?
    8.6 How committed are project members to defect data collection?
9. What are the defect fixing mechanisms?
    9.1 What information is relied on for fixing?
    9.2 What is the defect fixing strategy (for example fixing low severity defects later and attend to high severity defects now)?
10. What are the general practices at [company name] to make sure developers comply with practices and policies?
    10.1 Are there defined guidelines and procedures available to members of the project?
    10.2 Are there project specific standards that you have to comply with?
11. Is this a critical project in terms of reliability?
    11.1 What percentage of the components of the system is critical?
    11.2 Is there a difference between the way you handle critical components and non-critical ones in terms of development practices including requirements analysis, design, defect detection, fault reporting, etc.?
12. What are the mechanisms that ensure information flow between requirements analysis and testing?
    12.1 Are the defects detected traced back to test cases that detected them?

III. Detailed questions

13. How volatile are the requirements? How do you deal with changes during development in this project[3]?

[3] A change can be a requirement change, an enhancement request, refactoring, etc.

    13.1 Do you have mechanisms like a Change Control Board (CCB), league of experts or you use agile processes for this purpose?
    13.2 Are the defects traced back to requirements?
14. What are the communication mechanisms (Face-2-face, email, a proprietary system, Official meeting, unofficial meetings, Scrum standup meetings)?
    14.1 Is communication synchronic or is it deferred?
    14.2 How friendly is the interaction between project members; specifically testing staff and developers?
    14.3 How hard is it to organize a meeting in the project considering the busy schedules of parties involved?
15. What are the evolvability practices (commenting for code, coding standards, coding styles, design paradigm, etc.)?
    15.1 How much do you rely on them, for example on code comments in the project?
    15.2 Are there any coding standards defined for coders?
16. Is there a priority list or a similar mechanism to handle high priority tasks?
17. How often do the project members change? What about other staff members who have an influence on the project?
18. How much do you rely on and trust other project members? Is there a fear of data misuse by other members among project staff?
19. Please describe the office ergonomics.

APPENDIX 6 LITERATURE SOURCES FOR THE MAPPING STUDY

The list is presented in accordance with topic areas TABLE 2

Fault detection

Arthur, J. D., Gröner, M. K., Hayhurst, K. J., & Holloway, C. M. (1999). Evaluating the effectiveness of independent verification and validation. *Computer*, *32*(10), 79-83.

Aurum, A., Petersson, H., & Wohlin, C. (2002). State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, *12*(3), 133-154.

Basili, V. R., & Selby, R. W. (1987). Comparing the effectiveness of software testing strategies. *Software Engineering, IEEE Transactions on*, (12), 1278-1296.

Beer, A., & Peischl, B. (2011). Testing of Safety-Critical Systems–a Structural Approach to Test Case Design. In *Advances in Systems Safety* (pp. 187-211). Springer London.

Bjarnason, E., Runeson, P., Borg, M., Unterkalmsteiner, M., Engström, E., Regnell, B., ... & Feldt, R. (2014). Challenges and practices in aligning requirements with verification and validation: a case study of six companies. *Empirical Software Engineering*, *19*(6), 1809-1855.

Chang, J. R., Huang, C. Y., Hsu, C. J., & Tsai, T. H. (2012). Comparative performance evaluation of applying extended PIE technique to accelerate software testability analysis. *International Journal of Systems Science*, *43*(12), 2314-2333.

Chernak, Y. (1996). A statistical approach to the inspection checklist formal synthesis and improvement. *Software Engineering, IEEE Transactions on*, *22*(12), 866-874.

Cotroneo, D., Pietrantuono, R., & Russo, S. (2013). Testing techniques selection based on ODC fault types and software metrics. *Journal of Systems and Software*, *86*(6), 1613-1637.

Dhambri, K., Sahraoui, H., & Poulin, P. (2008, April). Visual detection of design anomalies. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on* (pp. 279-283). IEEE.

Fang, Q., Zhang, C., Ye, X., Shi, J., & Zhang, X. (2014, June). A new approach for developing safety-critical software in automotive industry. In *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on* (pp. 64-69). IEEE.

Felderer, M., & Beer, A. (2013). Using defect taxonomies to improve the maturity of the system test process: results from an industrial case study. In *software quality. Increasing value in software and systems development* (pp. 125-146). Springer Berlin Heidelberg.

Felderer, M., & Schieferdecker, I. (2014). A taxonomy of risk-based testing. *International Journal on Software Tools for Technology Transfer*, *16*(5), 559-568.

Fu, J., Lu, M., & Liu, B. (2009, December). Software Testability Measurement Based on Rough Set Theory. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on* (pp. 1-4). IEEE.

Gelperin, D., & Hetzel, B. (1988). The growth of software testing. *Communications of the ACM*, *31*(6), 687-695.

Grechanik, M., Jones, J. A., Orso, A., & van der Hoek, A. (2010, November). Bridging gaps between developers and testers in globally-distributed software development. In

Proceedings of the FSE/SDP workshop on Future of software engineering research (pp. 149-154). ACM.

Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. *ACM Sigplan Notices*, *39*(12), 92-106.

Janzen, D., & Saiedian, H. (2005). Test-driven development: Concepts, taxonomy, and future direction. *Computer*, (9), 43-50.

Joshi, M., & Sardana, N. (2014, March). Design and code time testability analysis for object oriented systems. In *Computing for Sustainable Global Development (INDIACom), 2014 International Conference on* (pp. 590-592). IEEE.

Kukkanen, J., Vakevainen, K., Kauppinen, M., & Uusitalo, E. (2009, December). Applying a systematic approach to link requirements and testing: a case study. In *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific* (pp. 482-488). IEEE.

Laitenberger, O. (1998, November). Studying the effects of code inspection and structural testing on software quality. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on* (pp. 237-246). IEEE.

Laitenberger, O., & DeBaud, J. M. (2000). An encompassing life cycle centric survey of software inspection. *Journal of systems and software*, *50*(1), 5-31.

Mäntylä, M. V., & Itkonen, J. (2014). How are software defects found? The role of implicit defect detection, individual responsibility, documents, and knowledge. *Information and Software Technology*, *56*(12), 1597-1612.

Mantyla, M. V., & Lassenius, C. (2009). What types of defects are really discovered in code reviews?. *Software Engineering, IEEE Transactions on*, *35*(3), 430-448.

Miller, S. P., Tribble, A. C., Whalen, M. W., & Heimdahl, M. P. (2006). Proving the shalls. *International Journal on Software Tools for Technology Transfer*, *8*(4-5), 303-319.

Musa, J. D. (1996). Software reliability-engineered testing. *Computer*, *29*(11), 61-68.

Porter, A. A., Siy, H. P., Toman, C. A., & Votta, L. G. (1997). An experiment to assess the cost-benefits of code inspections in large scale software development. *Software Engineering, IEEE Transactions on*, *23*(6), 329-346.

Post, H., Sinz, C., Merz, F., Gorges, T., & Kropf, T. (2009, August). Linking functional requirements and software verification. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International* (pp. 295-302). IEEE.

Pullum, L. L., & Dugan, J. B. (1996, January). Fault tree models for the analysis of complex computer-based systems. In *Reliability and Maintainability Symposium, 1996 Proceedings. International Symposium on Product Quality and Integrity., Annual* (pp. 200-207). IEEE.

Rafi, D. M., Moses, K. R. K., Petersen, K., & Mäntylä, M. V. (2012, June). Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test* (pp. 36-42). IEEE Press.

Runeson, P., Andersson, C., Thelin, T., Andrews, A., & Berling, T. (2006). What do we know about defect detection methods?. *IEEE software*, *23*(3), 82.

Siy, H., & Votta, L. (2001, November). Does the modern code inspection have value?. In *Proceedings of the IEEE international Conference on Software Maintenance (ICSM'01)* (p. 281). IEEE Computer Society.

Uusitalo, E. J., Komssi, M., Kauppinen, M., & Davis, A. M. (2008, September). Linking requirements and testing in practice. In *International Requirements Engineering, 2008. RE'08. 16th IEEE* (pp. 265-270). IEEE.

van Genuchten, M., Van Dijk, C., Scholten, H., & Vogel, D. (2001). Using group support systems for software inspections. *Software, IEEE*, *18*(3), 60-65.

Van Moll, J. H., Jacobs, J. C., Freimut, B., & Trienekens, J. J. M. (2002, October). The importance of life cycle modeling to defect detection and prevention. In *Software Technology and Engineering Practice, 2002. STEP 2002. Proceedings. 10th International Workshop on* (pp. 144-155). IEEE.

van Moll, J., Jacobs, J., Kusters, R., & Trienekens, J. (2004). Defect detection oriented lifecycle modeling in complex product development. *Information and Software Technology*, *46*(10), 665-675.

Vegas, S., Juristo, N., & Basili, V. (2006). Packaging experiences for improving testing technique selection. *Journal of Systems and Software*, *79*(11), 1606-1618.

Voas, J. M., & Miller, K. W. (1995). Software testability: The new verification. *IEEE software*, *12*(3), 17.

Vorobyov, K., & Krishnan, P. (2010). Comparing model checking and static program analysis: A case study in error detection approaches. *Proc. SSV*, 1-7.

Whittaker, J. A. (2000). What is software testing? And why is it so hard?. *Software, IEEE*, *17*(1), 70-79.

Wilkerson, J. W., Nunamaker Jr, J. F., & Mercer, R. (2012). Comparing the defect reduction benefits of code inspection and test-driven development. *Software Engineering, IEEE Transactions on*, *38*(3), 547-560.

Williams, L., Maximilien, E. M., & Vouk, M. (2003, November). Test-driven development as a defect-reduction practice. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on* (pp. 34-45). IEEE.

Wood, M., Roper, M., Brooks, A., & Miller, J. (1997, November). Comparing and combining software defect detection techniques: a replicated empirical study. In *ACM SIGSOFT Software Engineering Notes* (Vol. 22, No. 6, pp. 262-277). Springer-Verlag New York, Inc.

Xie, Y., & Engler, D. (2002). Using redundancies to find errors. *ACM SIGSOFT Software Engineering Notes*, *27*(6), 51-60.

Zelkowitz, M. V., & Rus, I. (2004). Defect evolution in a product line environment. *Journal of Systems and Software*, *70*(1), 143-154.

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P., & Vouk, M. A. (2006). On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*, *32*(4), 240-253.

Human factors

Amrit, C., Daneva, M., & Damian, D. (2014). Human factors in software development: On its underlying theories and the value of learning from related disciplines. A guest editorial introduction to the special issue. Information and Software Technology, 56(12), 1537-1542.

França, A. C. C., Da Silva, F. Q., de LC Felix, A., & Carneiro, D. E. (2014). Motivation in software engineering industrial practice: A cross-case analysis of two software organisations. Information and Software Technology, 56(1), 79-101.

Huang, F., Liu, B., & Huang, B. (2012). A taxonomy system to identify human error causes for software defects. In Proceedings 18th Issat International Conference on Reliability & Quality in Design, Boston, USA (pp. 44-49).

Huang, F., Liu, B., Song, Y., & Keyal, S. (2014). The links between human error diversity and software diversity: Implications for fault diversity seeking. Science of Computer Programming, 89, 350-373.

Spichkova, M., Liu, H., Laali, M., & Schmidt, H. W. (2015). Human factors in software reliability engineering. arXiv preprint arXiv:1503.03584.

Reliability modeling

Babu, P. A., Kumar, C. S., & Murali, N. (2012). A hybrid approach to quantify software reliability in nuclear safety systems. Annals of Nuclear Energy, 50, 133-140.

Banerjee, S., Srikanth, H., & Cukic, B. (2010, November). Log-based reliability analysis of software as a service (saas). In Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on (pp. 239-248). IEEE.

Bishop, P. (2013). Does Software Have to Be Ultra Reliable in Safety Critical Systems?. In Computer Safety, Reliability, and Security (pp. 118-129). Springer Berlin Heidelberg.

Butler, R. W., & Finelli, G. B. (1993). The infeasibility of quantifying the reliability of life-critical real-time software. Software Engineering, IEEE Transactions on, 19(1), 3-12.

Goel, A. L. (1985). Software reliability models: Assumptions, limitations, and applicability. Software Engineering, IEEE Transactions on, (12), 1411-1423.

Zahedi, F. (1987). Reliability of information systems based on the critical success factors-formulation. Mis Quarterly, 187-203.

Zhang, X., & Pham, H. (2000). An analysis of factors affecting software reliability. Journal of Systems and Software, 50(1), 43-56.

Fault reporting and RCA

Basili, V. R., & Rombach, H. D. (1987, March). Tailoring the software process to project goals and environments. In Proceedings of the 9th international conference on Software Engineering (pp. 345-357). IEEE Computer Society Press.

Bhandari, I., Halliday, M., Tarver, E., Brown, D., Chaar, J., & Chillarege, R. (1993). A case study of software process improvement during development. Software Engineering, IEEE Transactions on, 19(12), 1157-1170.

Børretzen, J. A. Software Fault Reporting Processes in Business-Critical Systems (Doctoral dissertation, Norwegian University for Science and Technology).

Bridge, N., & Miller, C. (1998). Orthogonal defect classification using defect data to improve software development. Software Quality, 3(1), 1-8.

Card, D. N. (1998). Learning from our mistakes with defect causal analysis. Software, IEEE, 15(1), 56-63.

Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., & Wong, M. Y. (1992). Orthogonal defect classification-a concept for in-process measurements. Software Engineering, IEEE Transactions on,18(11), 943-956.

El Emam, K., & Wieczorek, I. (1998, November). The repeatability of code defect classifications. In Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on (pp. 322-333). IEEE.

Freimut, B., Denger, C., & Ketterer, M. (2005, September). An industrial case study of implementing and validating defect classification for process improvement and quality management. In Software Metrics, 2005. 11th IEEE International Symposium (pp. 10-pp). IEEE.

Grady, R. B. (1996). Software failure analysis for high-return process improvement decisions. Hewlett Packard Journal, 47, 15-24.

Granda, M. F., Condori-Fernandez, N., Vos, T. E., & Pastor, O. (2015, May). What do we know about the defect types detected in conceptual models?. In Research Challenges in Information Science (RCIS), 2015 IEEE 9th International Conference on (pp. 88-99). IEEE.

Hayes, J. H., Raphael, I., Holbrook, E. A., & Pruett, D. M. (2006, August). A case history of International Space Station requirement faults. In Engineering of Complex Computer Systems, 2006. ICECCS 2006. 11th IEEE International Conference on (pp. 10-pp). IEEE.

Hong, G. Y., Xie, M., & Shanmugan, P. (1999). A statistical method for controlling software defect detection process. Computers & industrial engineering, 37(1), 137-140.

Huang, L., Ng, V., Persing, I., Chen, M., Li, Z., Geng, R., & Tian, J. (2015). AutoODC: Automated generation of orthogonal defect classifications. Automated Software Engineering, 22(1), 3-46.

Huber, J. T. (2000). A comparison of IBM's orthogonal defect classification to Hewlett Packard's defect origins, types, and modes. In Proceedings of International Conference on Applications of Software Measurement. San Jose, CA (pp. 1-17).

Jalote, P., & Agrawal, N. (2005, December). Using defect analysis feedback for improving quality and productivity in iterative software development. InInformation and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on (pp. 703-713). IEEE.

Kalinowski, M., Travassos, G. H., & Card, D. N. (2008, September). Towards a defect prevention based process improvement approach. In Software Engineering and Advanced Applications, 2008. SEAA'08. 34th Euromicro Conference (pp. 199-206). IEEE.

Kidwell, B., & Hayes, J. (2015, March). Toward a learned project-specific fault taxonomy: application of software analytics. In 2015 IEEE 1st International Workshop on Software Analytics (SWAN) (pp. 1-4). IEEE.

Lehtinen, T. O., Mäntylä, M. V., & Vanhanen, J. (2011). Development and evaluation of a lightweight root cause analysis method (ARCA method)–field studies at four software companies. Information and Software Technology,53(10), 1045-1061.

Leszak, M., Perry, D. E., & Stoll, D. (2002). Classification and evaluation of defects in a project retrospective. Journal of Systems and Software, 61(3), 173-187.

Lewis, N. D. (1999). Assessing the evidence from the use of SPC in monitoring, predicting & improving software quality. Computers & industrial engineering, 37(1), 157-160.

Li, N., Li, Z., & Sun, X. (2010, December). Classification of software defect detected by black-box testing: An empirical study. In Software Engineering (WCSE), 2010 Second World Congress on (Vol. 2, pp. 234-240). IEEE.

Ma, L., & Tian, J. (2007). Web error classification and analysis for reliability improvement. Journal of Systems and Software, 80(6), 795-804.

Margarido, I. L., Faria, J. P., Vidal, R. M., & Vieira, M. (2011, June). Classification of defect types in requirements specifications: Literature review, proposal and assessment. In

Information Systems and Technologies (CISTI), 2011 6th Iberian Conference on (pp. 1-6). IEEE.

Mellegard, N., Staron, M., & Torner, F. (2012, November). A light-weight defect classification scheme for embedded automotive software and its initial evaluation. In Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on (pp. 261-270). IEEE.

Ploski, J., Rohr, M., Schwenkenberg, P., & Hasselbring, W. (2007). Research issues in software fault categorization. ACM SIGSOFT Software Engineering Notes, 32(6), 6.

Raninen, A., Toroi, T., Vainio, H., & Ahonen, J. J. (2012). Defect data analysis as input for software process improvement. In Product-Focused Software Process Improvement (pp. 3-16). Springer Berlin Heidelberg.

Shenvi, A. A. (2009, February). Defect prevention with orthogonal defect classification. In Proceedings of the 2nd India software engineering conference (pp. 83-88). ACM.

Thung, F., Lo, D., & Jiang, L. (2012, October). Automatic defect categorization. In Reverse Engineering (WCRE), 2012 19th Working Conference on (pp. 205-214). IEEE.

Vallespir, D., Grazioli, F., & Herbert, J. (2009). A framework to evaluate defect taxonomies. In XV Congreso Argentino de Ciencias de la Computación.

Wagner, S. (2008, July). Defect classification and defect types revisited. In Proceedings of the 2008 workshop on Defects in large software systems (pp. 39-40). ACM.

Yu, W. D. (1998). A software fault prevention approach in coding and root cause analysis. Bell Labs Technical Journal, 3(2), 3-21.

Agile

Arnold, R. S. (1989). Software restructuring. Proceedings of the IEEE, 77(4), 607-617.

Bowers, J., May, J., Melander, E., Baarman, M., & Ayoob, A. (2002). Tailoring XP for large system mission critical software development. In Extreme Programming and Agile Methods—XP/Agile Universe 2002 (pp. 100-111). Springer Berlin Heidelberg.

Cao, L., & Ramesh, B. (2008). Agile requirements engineering practices: An empirical study. Software, IEEE, 25(1), 60-67.

Carpenter, S. E., & Dagnino, A. (2014, September). Is Agile too Fragile for Space-Based Systems Engineering?. In Space Mission Challenges for Information Technology (SMC-IT), 2014 IEEE International Conference on (pp. 38-45). IEEE.

Gary, K., Enquobahrie, A., Ibanez, L., Cheng, P., Yaniv, Z., Cleary, K., ... & Heidenreich, J. (2011). Agile methods for open source safety-critical software. Software: Practice and Experience, 41(9), 945-962.

Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., ... & Kähkönen, T. (2004). Agile software development in large organizations. Computer, 37(12), 26-34.

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. Software Engineering, IEEE Transactions on, 30(2), 126-139.

Rech, J. (2007). Handling of software quality defects in agile software development. Agile Software Development Quality Assurance, 90.

Sidky, A., & Arthur, J. (2007, March). Determining the applicability of agile practices to mission and life-critical systems. In Software Engineering Workshop, 2007. SEW 2007. 31st IEEE (pp. 3-12). IEEE.

Fault prediction

Canfora, G., & Cerulo, L. (2005, September). Impact analysis by mining software and change request repositories. In Software Metrics, 2005. 11th IEEE International Symposium (pp. 9-pp). IEEE.

Catal, C., & Diri, B. (2009). A systematic review of software fault prediction studies. Expert systems with applications, 36(4), 7346-7354.

Chaturvedi, K. K., & Singh, V. B. (2012, September). Determining bug severity using machine learning techniques. In Software Engineering (CONSEG), 2012 CSI Sixth International Conference on (pp. 1-6). IEEE.

Eick, S. G., Loader, C. R., Long, M. D., Votta, L. G., & Vander Wiel, S. (1992, June). Estimating software fault content before coding. In Proceedings of the 14th international conference on Software engineering (pp. 59-65). ACM.

Fenton, N. E., & Neil, M. (1999). A critique of software defect prediction models. Software Engineering, IEEE Transactions on, 25(5), 675-689.

Fenton, N. E., & Ohlsson, N. (2000). Quantitative analysis of faults and failures in a complex software system. Software Engineering, IEEE Transactions on,26(8), 797-814.

Fluri, B., & Gall, H. C. (2006, June). Classifying change types for qualifying change couplings. In Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on (pp. 35-45). IEEE.

Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. (2000). Predicting fault incidence using software change history. Software Engineering, IEEE Transactions on, 26(7), 653-661.

Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. Software Engineering, IEEE Transactions on, 38(6), 1276-1304.

Herrmann, D. S. (1998, January). Sample implementation of the Littlewood holistic model for assessing software quality, safety and reliability. In Reliability and Maintainability Symposium, 1998. Proceedings., Annual (pp. 138-148). IEEE.

Jacobs, J., Van Moll, J., Krause, P., Kusters, R., Trienekens, J., & Brombacher, A. (2005). Exploring defect causes in products developed by virtual teams. Information and Software Technology, 47(6), 399-410.

Khoshgoftaar, T. M., & Seliya, N. (2004). Comparative assessment of software quality classification techniques: An empirical case study. Empirical Software Engineering, 9(3), 229-257.

Kidwell, B., Hayes, J. H., & Nikora, A. P. (2014, October). Toward Extended Change Types for Analyzing Software Faults. In Quality Software (QSIC), 2014 14th International Conference on (pp. 202-211). IEEE.

Kim, S., Zimmermann, T., Pan, K., & Whitehead Jr, E. J. (2006, September). Automatic identification of bug-introducing changes. In Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on (pp. 81-90). IEEE.

Li, B., Sun, X., Leung, H., & Zhang, S. (2013). A survey of code-based change impact analysis techniques. Software Testing, Verification and Reliability, 23(8), 613-646.

Lutz, R. R., & Mikulski, I. C. (2004). Empirical analysis of safety-critical anomalies during operations. Software Engineering, IEEE Transactions on, 30(3), 172-180.

Munson, J. C., & Khoshgoftaar, T. M. (1992). The detection of fault-prone programs. Software Engineering, IEEE Transactions on, 18(5), 423-433.

Nagappan, N., & Ball, T. (2005, May). Use of relative code churn measures to predict system defect density. In Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on (pp. 284-292). IEEE.

Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005). Predicting the location and number of faults in large software systems. Software Engineering, IEEE Transactions on, 31(4), 340-355.

Rana, R., Staron, M., Hansson, J., Nilsson, M., & Meding, W. (2014, August). A framework for adoption of machine learning in industry for software defect prediction. In Software Engineering and Applications (ICSOFT-EA), 2014 9th International Conference on (pp. 383-392). IEEE.

Rapu, D., Ducasse, S., Gîrba, T., & Marinescu, R. (2004, March). Using history information to improve design flaws detection. In Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on (pp. 223-232). IEEE.

Shen, V. Y., Yu, T. J., Thebaut, S. M., & Paulsen, L. R. (1985). Identifying error-prone software—an empirical study. Software Engineering, IEEE Transactions on, (4), 317-324.

Vander Wiel, S. A., & Votta, L. G. (1993). Assessing software designs using capture-recapture methods. Software Engineering, IEEE Transactions on, 19(11), 1045-1054.

Zhou, Y., & Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. Software Engineering, IEEE Transactions on, 32(10), 771-789.

Safety

Alemzadeh, H., Iyer, R. K., Kalbarczyk, Z., & Raman, J. (2013). Analysis of safety-critical computer failures in medical devices. Security & Privacy, IEEE, 11(4), 14-26.

Dunn, W. R. (2004). Software safety and reliability.

Favarò, F. M., Jackson, D. W., Saleh, J. H., & Mavris, D. N. (2013). Software contributions to aircraft adverse events: Case studies and analyses of recurrent accident patterns and failure mechanisms. Reliability Engineering & System Safety, 113, 131-142.

Herrmann, D. S., & Peercy, D. E. (1999, January). Software reliability cases: the bridge between hardware, software and system safety and reliability. In Reliability and Maintainability Symposium, 1999. Proceedings. Annual (pp. 396-402). IEEE.

Ibrahim, W. M., Bettenburg, N., Adams, B., & Hassan, A. E. (2012). On the relationship between comment update practices and software bugs. Journal of Systems and Software, 85(10), 2293-2304.

Ishimatsu, T., Leveson, N. G., Thomas, J. P., Fleming, C. H., Katahira, M., Miyamoto, Y., ... & Hoshino, N. (2014). Hazard analysis of complex spacecraft using systems-theoretic process analysis. Journal of Spacecraft and Rockets, 51(2), 509-522.

Leveson, N. G., & Turner, C. S. (1993). An investigation of the Therac-25 accidents. Computer, 26(7), 18-41.

maintenance

Mathur, S., & Malik, S. (2010). Advancements in the V-Model. International Journal of Computer Applications, 1(12).

Wears, R. L., & Leveson, N. G. (2008). Safeware": safety-critical computing and healthcare information technology. Advances in patient safety: new directions and alternative approaches, 4, 1-10.

Defect analysis

Damm, L. O., Lundberg, L., & Wohlin, C. (2004). Determining the improvement potential of a software development organization through fault analysis: a method and a case study. In Software Process Improvement (pp. 138-149). Springer Berlin Heidelberg.

Hamill, M., & Goseva-Popstojanova, K. (2009). Common trends in software fault and failure data. Software Engineering, IEEE Transactions on, 35(4), 484-496.

Lange, C. F., & Chaudron, M. R. (2006, May). Effects of defects in UML models: an experimental investigation. In Proceedings of the 28th international conference on Software engineering (pp. 401-411). ACM.

Moha, N., Gueheneuc, Y. G., Duchien, L., & Le Meur, A. F. (2010). DECOR: A method for the specification and detection of code and design smells. Software Engineering, IEEE Transactions on, 36(1), 20-36.

Nugroho, A., & Chaudron, M. R. (2014). The impact of UML modeling on defect density and defect resolution time in a proprietary system. Empirical Software Engineering, 19(4), 926-954.

Ott, D. (2012, September). Defects in natural language requirement specifications at mercedes-benz: An investigation using a combination of legacy data and expert opinion. In Requirements Engineering Conference (RE), 2012 20th IEEE International (pp. 291-296). IEEE.

Pan, K., Kim, S., & Whitehead Jr, E. J. (2009). Toward an understanding of bug fix patterns. Empirical Software Engineering, 14(3), 286-315.

Fault reduction

Alho, P., & Mattila, J. (2011). Dependable control systems design and evaluation. In Conference on Systems Engineering Research (CSER) 2011.

Asthana, A., & Okumoto, K. (2012). Integrative Software Design for Reliability: Beyond Models and Defect Prediction. Bell Labs Technical Journal, 17(3), 37-59.

Boehm, B. W., Mcclean, R. K., & Urfrig, D. E. (1975). Some experience with automated aids to the design of large-scale reliable software. Software Engineering, IEEE Transactions on, (1), 125-133.

Boehm, B., & Basili, V. R. (2005). Software defect reduction top 10 list. Foundations of empirical software engineering: the legacy of Victor R. Basili, 426.

Carrozza, G., Pietrantuono, R., & Russo, S. (2015). Defect analysis in mission-critical software systems: a detailed investigation. Journal of Software: Evolution and Process, 27(1), 22-49.

Dutertre, B., & Stavridou, V. (1997). Formal requirements analysis of an avionics control system. Software Engineering, IEEE Transactions on, 23(5), 267-278.

Hayes, J. H., Chemannoor, I. R., & Holbrook, E. A. (2011). Improved code defect detection with fault links. Software Testing, Verification and Reliability, 21(4), 299-325.

Heimdahl, M. P., & Heitmeyer, C. L. (1998). Formal methods for developing high assurance computer systems: Working group report. In Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on (pp. 60-64). IEEE.

Jacobs, J., Van Moll, J., Kusters, R., Trienekens, J., & Brombacher, A. (2007). Identification of factors that influence defect injection and detection in development of software intensive products. Information and Software Technology, 49(7), 774-789.

Johnson, P. M., Kou, H., Agustin, J., Chan, C., Moore, C., Miglani, J., ... & Doane, W. E. (2003, May). Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In Proceedings of the 25th international Conference on Software Engineering (pp. 641-646). IEEE Computer Society.

Selby, R. W., Basili, V. R., & Baker, F. T. (1987). Cleanroom software development: an empirical evaluation. Software Engineering, IEEE Transactions on, (9), 1027-1037.

Shull, F., Basili, V., Boehm, B., Brown, A. W., Costa, P., Lindvall, M., ... & Zelkowitz, M. (2002). What we have learned about fighting defects. In Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on (pp. 249-258). IEEE.

Walia, G. S., & Carver, J. C. (2013). Using error abstraction and classification to improve requirement quality: conclusions from a family of four empirical studies. Empirical Software Engineering, 18(4), 625-658.

Zhang, X., Stafford, T. F., Dhaliwal, J. S., Gillenson, M. L., & Moeller, G. (2014). Sources of conflict between developers and testers in software development. Information & Management, 51(1), 13-26.

Lanubile, F., Shull, F., & Basili, V. R. (1998, November). Experimenting with error abstraction in requirements documents. In Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International (pp. 114-121). IEEE.

Stavely, A. M. (1999, March). High-quality software through semiformal specification and verification. In Software Engineering Education and Training, 1999. Proceedings. 12th Conference on (pp. 145-155). IEEE.

Walia, G. S., & Carver, J. C. (2013, November). Using error information to improve software quality. In Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on (pp. 107-107). IEEE.

AlShathry, O. (2014, March). Operational profile modeling as a risk assessment tool for software quality techniques. In Computational Science and Computational Intelligence (CSCI), 2014 International Conference on (Vol. 2, pp. 181-184). IEEE.

Process improvement

Harter, D. E., Kemerer, C. F., & Slaughter, S. A. (2012). Does software process improvement reduce the severity of defects? A longitudinal field study. Software Engineering, IEEE Transactions on, 38(4), 810-827.

Huang, F., Liu, B., Wang, S., & Li, Q. (2015). The impact of software process consistency on residual defects. Journal of Software: Evolution and Process, 27(9), 625-646.

Lohmann, N. (2013). Compliance by design for artifact-centric business processes. Information Systems, 38(4), 606-618.

Tools

Cousot, P. (2007, September). Proving the absence of run-time errors in safety-critical avionics code. In Proceedings of the 7th ACM & IEEE international conference on Embedded software (pp. 7-9). ACM.

Hallem, S., Chelf, B., Xie, Y., & Engler, D. (2002). A system and language for building system-specific, static analyses (Vol. 37, No. 5, pp. 69-82). ACM.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013, May). Why don't software developers use static analysis tools to find bugs?. In Software Engineering (ICSE), 2013 35th International Conference on (pp. 672-681). IEEE.

Novak, J., Krajnc, A., & Zontar, R. (2010, May). Taxonomy of static code analysis tools. In MIPRO, 2010 Proceedings of the 33rd International Convention (pp. 418-422). IEEE.

Wassyng, A., & Lawford, M. (2006). Software tools for safety-critical software development. International Journal on Software Tools for Technology Transfer, 8(4-5), 337-354.

Software reliability engineering

Musa, J. D., & Everett, W. W. (1990). Software-reliability engineering: Technology for the 1990s. Software, IEEE, 7(6), 36-43.

Herrmann, D. S. (1996). A methodology for evaluating, comparing, and selecting software safety and reliability standards. Aerospace and Electronic Systems Magazine, IEEE, 11(1), 3-12.

Hanmer, R. S., McBride, D. T., & Mendiratta, V. B. (2007). Comparing reliability and security: Concepts, requirements, and techniques. Bell Labs Technical Journal, 12(3), 65-78.

Carman, D. W., Dolinsky, A. A., Lyu, M. R., & Yu, J. S. (1995, October). Software reliability engineering study of a large-scale telecommunications software system. In Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on (pp. 350-359). IEEE.

Vouk, M. A. (2000, January). Software reliability engineering. In a tutorial presented at the Annual Reliability and Maintainability Symposium http://renoir. csc. ncsu. edu/Faculty/Vouk/vouk_se. html.

Lyu, M. R. (2007, May). Software reliability engineering: A roadmap. In 2007 Future of Software Engineering (pp. 153-170). IEEE Computer Society.