Studying Things that Go Wrong in Software Development

Tamara Lopez

09 February, 2012

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk
STUDYING THINGS THAT GO WRONG
IN SOFTWARE DEVELOPMENT

Author: Tamara Lopez

Supervisors: Prof. Marian Petre
Prof. Bashar Nuseibeh

December 13, 2011
1. Introduction

Dependability (see also Appendix A for A Note About the Language of Error) has been a theme in software engineering research at least since its emergence as a key problem in computing at the NATO software engineering conferences (Naur and Randell, 1969; Buxton and Randell, 1970). Despite a great deal of effort in the intervening decades, the threat of software failure and the quest for a multivalent yet comprehensive sense of what dependable software is remain powerful drivers for research and provocative tropes in anecdotal accounts of computing.

Anecdotal accounts offer provocative reasons for poor dependability in software. In *The Mythical Man Month* Brooks claimed that because the medium of software is built out of "thought-stuff" (p.7), the software industry is overly optimistic about the ease with which it is built, and unprepared to deal with the fact that faulty ideas lead to bugs in software. He maintained that conceptual integrity, the key to building successful software, is elusive: large software initiatives struggle to create something that is designed by many minds, but that remains conceptually coherent to individual users (Brooks, 1995). In *Engineering and the Mind's Eye*, Ferguson noted that complex software contains a "thousand points of doubt" and cautioned engineers to be vigilant in questioning the assumptions made by software developers who build computer-aided design software (Ferguson, 1992, p.183).

These sources suggest that software dependability is achieved or not as a result of decisions and judgements made by people, working alone and together. However they cannot explain exactly how this happens. Is it true that faulty ideas lead to bugs? How does this happen? What do faulty ideas look like? Points of doubt?

The research proposed here lies somewhere between the broad statements of anecdotal accounts and the narrowly focused goals of software engineering research in areas related to dependability. It will consider the cognitive and social factors that anecdotes suggest contribute to failure prone software. Like other empirical studies in software engineering, this research will develop a body of detailed evidence that may be used to improve software engineering practice. However, in contrast to both of these approaches, this research will trace specific, mistaken choices made within software initiatives. Similarly, it will address gaps in the empirical data by examining these mistakes in the context of work as it happens, and by focusing on the thoughts and behaviour of the people involved in software development.

2. Research Question

In the introduction to his seminal root-cause analysis, Endres commented:
Almost everyone who has ever written a program that did not immediately function as intended - a normal occurrence as we all know - has developed his personal theory about what went wrong in this specific case and why. As a result, the programming style is modified the next time, i.e. the tricks which were unsuccessful are avoided, or more attention is directed to typically error-prone areas. It would be desirable, of course, that this learning process, which is individually experienced by a competent programmer, be expanded to include a larger group of programmers, or even the entire profession. (Endres, 1975, p. 327)

This research shares Endres’ desire: to understand the personal theories developers have about things that go wrong while making software, to record the character of individual mistakes, to catalog specific instances, and to document the strategies employed to deal with them. It takes as its starting point the following research question:

How do developers understand, manage and communicate things that go wrong while making software?

Developer used here is broadly conceived, referring to people who are involved in making software, for example (but not exclusively): coders, analysts, systems administrators, and designers. Similarly, making software is meant to indicate the day-to-day work undertaken in software engineering projects with two wholly pragmatic exceptions: it will not examine the process of agreeing requirements with stakeholders, though it will consider how the results of these activities are utilised by developers. It also will not examine the problems that arise when software is in use except as they are reported back to the development team.

So conceived, making software includes activities that correspond to established software engineering methodologies and organisational process, but also encompasses the smaller, informal actions performed in the course of development work. As in other design intensive disciplines, making software in this sense involves collective and individual effort to identify and structure problems related to software development and to formulate appropriate solutions (Cross, 2001) to these problems. It pays particular attention to the makers and making of artefacts, and not exclusively to the way that software artefacts change along the way (Curtis et al., 1988).

The things that go wrong while making software are those issues encountered by developers in understanding what is to be done and deciding how best to go about doing it. Things that go wrong may result at some point in faults within source-code (Avižienis et al., 2004), or may manifest as inconsistencies (Nuseibeh et al., 2001) within other descriptions such as specifications and diagrams. Details and theories about what
went wrong may be included after the fact in the information reported within bug or modification requests (Aranda and Venolia, 2009), or may otherwise be communicated among team members. That is to say, evidence of things that go wrong may end up within the many artefacts that comprise the software record for a project and have measurable impact on the resulting software.

Goings wrong may also be those problems of which individual developers or groupings of developers are “secretly” aware (Endres, 1975, p. 330), and thus may also be ephemeral in at least two senses. First, they may originate only as mis-communications, mis-understandings or mis-perceptions that may accumulate and contribute over time to faults. Secondly, they may go wrong and be put right again in the course of daily work practice. In both cases, they are temporally defined, leaving no clear (or indeed any) representation within software artefacts. Instead they form lacunae, gaps in the software record that become increasingly difficult to understand or even to remember as time passes and the making moves on (Aranda and Venolia, 2009).

If they are put right, forgotten, or otherwise fall from view why do the things that go wrong matter? As the anecdotal accounts suggest, the software engineering research community wonders about them, and in particular about the effects they have on the ability of software engineering to achieve the highest level of dependability possible. Findings in the computing literature suggest several areas in which an investigation of things that go wrong may also yield new understanding for software engineering research and suggestions for improving software practice:

(1) What does an examination of development in practice reveal about the nature and characteristics of things that go wrong? Imprecise requirements and poor design are often reported after the fact as a cause for faults (Basili and Perricone, 1984; Perry and Stieg, 1993). Do things go wrong predominantly when developers are trying to understand what is required, or when they are deciding on solutions to meet requirements (Guindon, 1990)? What is the relationship of things that go wrong in individual work and the ways these are managed within informal (Ko et al., 2007) and formal group work (Seaman and Basili, 1997)?

(2) What factors influence and shape things that go wrong? How do knowledge and experience among developers correlate to things that go wrong (Perry and Stieg, 1993; Leszak et al., 2002)? How does design intent factor into the work that follows to make software (Endres, 1975; Ko et al., 2007)?

(3) How do things that go wrong influence and shape software? Interfaces have been shown to have a high incidence of faults (Basili and Perricone, 1984; Perry and
Evangelist, 1985, 1987) and both to facilitate collaboration and isolate developers (de Souza et al., 2004). What role do things that go wrong play in the development of such individual software features?

(4) What do things that go wrong reveal about the environment in which they are created? What can a study of things that go wrong reveal about individual cultures of development, software engineering practices, or model of design (Curtis et al., 1988)?

3. Literature Review

As noted in the introduction, the provision of dependable software has been a core theme in software engineering research for decades. One strand of this research analyses software that fails, while a second develops and tests techniques for ensuring that software doesn’t fail. A brief characterisation of these two strands as System and Dependability analyses follows. For a fuller treatment of these strands of research that includes references to representative studies, see also Appendix B: Perspectives on Failure Research in Computing.

System analyses identify weak elements in complex organisational, operational and software systems. Within them, individual or multiple faults become active at a moment in time or within a clearly bounded interval of time, and result in catastrophic or spectacular operational failure. Alternatively, software deemed to be “good enough” is released into production with significant problems that require costly maintenance, redesign and redevelopment. Systems analyses produce case studies and often do not conclude with specific, precise reasons for failure. Instead, they retrospectively identify the system or subsystem that failed, and provide general recommendations for improvement going forward. Even when they do isolate weaknesses in the processes of software creation or in particular software components, they do not produce general frameworks or models that can be extended to improve software engineering practice.

By contrast, dependability analyses treat smaller aspects or attributes of software engineering as they contribute to the goal of creating dependable software and thus to ensure that operational failure never occurs. These studies develop new or test existing techniques designed to make software ”better” (Perry, 2010, p.453). They employ a range of methods, often with the aim of examining a single part of the development process, and with a corresponding focus on achieving a single dependability mean. The studies are empirical in the sense that they work upon existing bodies of software, but they often employ quantitative analytical techniques, reducing natural language concepts related to software dependability (see also Appendix A for a description of these concepts) into
terms that can be measured, and thus used to demonstrate, verify and validate that software meets a quantifiable, pre-determined degree of dependability.

The research proposed here is not directly concerned with software that fails. Like studies that focus on achieving and improving software dependability, it explores smaller mistakes as they might hinder or support efforts made by developers to prevent failure.

The following review focuses on two branches of research that work to improve dependability in software from two perspectives. As the name suggests, root-cause analysis studies look for the sources of faults in software with an aim toward preventing faults. Coordination and cooperation studies support fault removal by examining how people coordinate their activities while fixing or maintaining complex software systems, and examine how tools mediate this work. In some ways, these branches of research are very different from one another: one works to improve process, the other to improve tools; one focuses primarily on the technical causes for errors and the other on their management; one uses a research model followed for some forty years within software engineering research, while the other is newer and less cohesive in approach and method. However different, this review will show that taken together, they offer the potential for a richer understanding of what errors in software are, and richer means for their study.

Since their earliest days, root-cause analyses have acknowledged simplification in defining what errors are, a compromise accepted in the effort to produce measurable improvement. They have done this at a cost, struggling in the intervening years to adequately or satisfactorily answer the question of why some errors occur, and limited by a definition of error that cannot account for non-quantifiable factors such as the workings of time and of humans. By contrast, the coordination and cooperation analyses focus specifically on both, examining sequences of events and the work of humans to remove faults from software. They demonstrate that improved understanding in these areas has important implications for supporting software development by uncovering weaknesses and gaps in how the software record is currently maintained. However like root-cause analyses, these studies depend upon a conceptualisation of error that equates the things that go wrong with faults as recorded in a piece of software. They cannot account for things that go wrong at other points in the process, which have to do with developer’s understanding of problems and how to solve them, and which may not leave evidence in the software record.

3.1. **Root-cause Analyses.** As noted, these studies aim to identify the source of bugs within development initiatives. Studies examine all phases of development including initial design, implementation, evolution and maintenance. Analysis draws primarily upon data collected from change reports or modification requests with the aim to make
software better: to reduce critical defects, to reduce the cost of developing software, and to find ways to improve the process of building software.

This section of the review takes the following form. First a model root-cause analysis is described in detail. Next, details of subsequent root-cause studies which have followed this model are given. Finally gaps in the research model are identified through comparative analysis of all the studies, and implications for future research are discussed.

3.1.1. Establishing a Model for Analysing the Root Causes of Errors. Albert Endres performed one early, influential root-cause analysis of software written for IBM in 1975 (Endres, 1975). The report is notable in that it has two complementary but different aims. Its first and perhaps principal aim is to establish a root-cause taxonomy for errors sampled from a five-month period of testing of release 28 of the operating system DOS/VS. The second aim of the paper is to use the root-cause analysis as a source for meditation about the nature of errors in software programming, and reflection about how they can be studied.

In answer to the first aim, a study was designed to plot error distribution and frequency in systems programming, software characterised by the author as beginning with "high quality" requirements but which structurally degrades over time due to its "growth pattern" (p. 327). The study examined failed test cases written to test the operating system post-integration, when the system had reached a basic state of operability. Two sets of test cases were run, one a series of regression tests to ensure that old functionality had not been compromised by new development, and one to simulate user inputs to the system. The test cases resulted in seven-hundred forty errors which were categorised by the original development team according to the error protocol which should be followed in their correction. Four-hundred thirty-two were deemed to be program errors – and thus not duplicates, documentation errors, hardware failures, operator errors or feature requests. These formed the data for analysis.

Analysis was performed to answer seven questions about each error:

- Where was the error made?
- When was the error made?
- Who made the error?
- What was done wrong?
- Why was the particular error made?
- What could have been done to prevent the error?
- Barring prevention, what could be done to detect the error?

Several assumptions were made. First, it was accepted that the actual error was defined according to the correction made, leaving aside the possibility that the correction addressed a symptom of a different, deeper problem. In addition, analysis equated the
number of errors with those resulting in failed test cases, and did not consider other problems that might be found and corrected along the way, or those which the programmer may have "secretly been aware of for some time" (p. 330). Additionally, though the information provided with failed test cases was sufficient to explain where and when the error was made, additional external information was required to answer which programmer made the error, necessarily gathered from information other than the test case data including conversation with the development team.

A taxonomy of error distribution by type of error was the primary outcome of analysis. This taxonomy included three main groups: one to categorise errors related to problem understanding (Group A in Appendix A below), one related to implementation (Group B) and one related to mechanical errors such as spelling, or errors in integrating modules (Group C). The taxonomy included a breakdown of factors contributing to the errors for several subgroups in classes A and B, and offered descriptive statistics for future detection of errors in classes A and B.

The study revealed that almost half of the errors examined fell into class A (46%), that is in the area of "understanding the problem, of problem communication, of the knowledge of possibilities and procedures for problem solving" (p.331). The reason for this was attributed by Endres to the complexity of the work, that is that the problems of systems programming are inherently ill-formed, dynamic and require iterative changes. He suggested that functional demands on the system can only be properly understood when they are seen in use, and suggested that to reduce the number of errors in this class, changes be made to development process, including the use of design and code walkthroughs, prototyping and user tests. Roughly the other half of the errors were attributed to programming technique, with suggestions that better programming methods including formal methods would reduce the number of errors.

In reflecting on the method of analysis, Endres argued that to truly understand why errors are made, one must remember that programming is a human activity. Comments supporting this view are sprinkled throughout the text, reflecting on the inner life of the programmer, on his motivations and his mental processes while programming, and suggesting the existence of personal strategies for managing the intellectual work of programming. Endres is a thorough analyst, and this reflection is synthesised into a categorisation of human causes of error in the Causes and Prevention of Errors section of the paper:

- technological (definability of the problem, feasibility of solving it, available procedures and tools),
- organisational (division of work load, available information, communication, resources),
• historic (history of the project, of the program, special situations, and external influences),
• group dynamic (willingness to cooperate, distribution of roles inside the project group),
• individual (experience, talent, and constitution of the individual programmer), and
• other (inexplicable causes) (Endres, 1975, p. 331)

He states that human-caused errors largely fall into those categories at the bottom of the list, within the "psychological" (p.331) realm, but concedes that of necessity, his own analysis adopts the much narrower and stricter view that errors are caused by a discrepancy between the difficulty of the problem and the adequacy of the means used to address it, and that their prevention is achieved by measures used to reduce that discrepancy. He suggests that though limited, this view results in constructive findings that can be addressed through technological and organisational means.

However perhaps in part as a result of this second current within the paper, the findings of the root-cause analysis are carefully qualified in other ways. Endres notes limitations to the approach, in particular that the errors analysed were reported in the final stage of testing, in the form of failed test cases. Thus, errors that would appear in early stages of a project, with less experienced programmers, or after a "hectic period of changes" were not well represented in the sample (p.328). He admits that sometimes errors lie "too deep" to be practically solved, and thus corrections may address consequences and not actual problems. He suggests that to really understand the roots of problems, it might be necessary not to examine the corrections made to software in the process of removing faults, but instead to compare the "intended implementation and the implementation actually carried out." (p. 329). In his remarks on causes and prevention of errors, he finds that no single "cure-all" exists for errors, but that each error has multiple causes and multiple means of prevention, and notes that given the many levels of error causes not considered in the analysis, the results are "sobering" (p. 332). These comments suggest that though the constraints of the study do result in constructive recommendations for reducing errors and improving practice, they do not ultimately succeed in generating a satisfactory answer to the question Why?, and perhaps also do not permit the author to adequately generalise the experience accumulated by individual programmers.

Nonetheless, the paper serves as a good model for reducing the concept of error into measures and means for study, a model whose basic structure is seen in many subsequent reports. The characteristics of six such studies are profiled and reviewed in the text that follows (see also Appendix B).
3.1.2. **Following the Research Model.** Fundamentally, studies after Endres’ accept his assertion that errors are constructively studied according to their technological and organisational causes and means of prevention. Following his model, taxonomies that represent the root-causes for errors form the centrepiece of analysis. They use data from a variety of sources: user and tester created bug and modification reports (Perry and Evangelist, 1985, 1987; Leszak et al., 2002), in-process questionnaires (Basili and Perricone, 1984) and retrospectively administered surveys (Perry and Stieg, 1993). In one case, the study design is experimental and the software examined is purpose-built (Schneidewind and Hoffmann, 1979), while in all other cases, empirical studies are made of large-scale aerospace and telecommunications software written in a variety of languages for a variety of operating environments. In the empirical studies, research is management sanctioned, with aims to improve practice and reduce costs associated with development. This has the consequence that constraints are placed on data collection and analysis methods and in one case, level of reporting detail and ability to validate results (Perry and Stieg, 1993).

The error taxonomies differ considerably in design, use and character. Some are designed *a priori*, by the researchers working alone (Schneidewind and Hoffmann, 1979; Basili and Perricone, 1984) or in collaboration with members of the development team (Perry and Stieg, 1993). Others are developed from analysis of the error data (Perry and Evangelist, 1985, 1987). In one case, a previously created taxonomy (Perry and Evangelist, 1985, i.e. the one developed in) is adapted and developed to represent additional information, and to improve usability (Leszak et al., 2002). The function the taxonomies serve in studies is also different. Basili and Perricone (1984) include the classification in a change report form (see Appendix C) completed by programmers. Likewise, Perry and Stieg (1993) surveyed programmers responsible for closing modification reports asking them to classify the error into one of nine fault type categories, and to indicate the phase of testing in which the error emerged.

However populated, the classified body of errors forms the basis for additional examination of particular code features such as complexity (Schneidewind and Hoffmann, 1979), interface defects (Perry and Evangelist, 1985, 1987) or more generally, environmental factors that influence software dependability (Basili and Perricone, 1984). The subsequent analysis may be reported in the form of narrative description, with additional groupings of the errors according to different conceptual parameters, and may include the use of and descriptive or inferential statistics.

The discussion and findings that follow from analysis of the error classifications in the papers similarly address familiar software engineering themes. Complexity is found both to correlate to error frequency (Schneidewind and Hoffmann, 1979), and not to (Basili
and Perricone, 1984). Application programming interfaces are found to have particularly high frequencies of errors associated with them (Perry and Evangelist, 1985, 1987) while these and other causes are evaluated in terms of the costs associated with their finding and fixing (Schneidewind and Hoffmann, 1979; Basili and Perricone, 1984; Leszak et al., 2002). As with reflective accounts, these studies also report problems in maintaining conceptual integrity or coherence, a common though less clearly stated theme in the findings.

For Endres, difficulties in maintaining conceptual integrity fall into the class he describes as related to problems of understanding (see Category A in Appendix A). Other researchers tend to conflate a notion of problem understanding with constructs drawn from software engineering process. For example, Basili and Perricone find that roughly half of all errors are in the area of requirement and functional specifications (1984). Perry and Evangelist note that the project they study included phases of "high-level design" and "detailed design" (Section 2 Background for the study) but find somewhat ambiguously that only a quarter of the interface errors they study are "design" errors. Their concession that "cryptic design requirements" (Section 4 Summary of Initial Results) could result in errors being incorrectly classified as implementation errors suggest that they are only considering "detailed" design in their analysis, while at the same time conceding that it may be skewing their results (1987).

In some cases, problems in maintaining conceptual integrity are reflected in the terms used within the root-cause taxonomies. Perry and Steig designed a second survey for their case study that included a section for identifying the “underlying causes” of design and coding errors. All of the members this category, examples of which include Ambiguous design and Knowledge incomplete (Section 5.1 Questionnaire) might be considered to represent difficulties related to maintaining conceptual integrity (1993). Indeed, their analysis found that lack of information dominated the underlying causes of the errors, while knowledge intensive activities such as code inspections dominated the means of prevention. Likewise Leszak et. al, in categories within their defect types defect types and human related triggers dimensions of root-causes for errors, acknowledge that a full understanding of why errors are made must include some information about human understanding – where it is lacking, how it is coordinated and maintained. Their findings confirm Perry and Steig’s conclusion that knowledge is one of the largest problems in software development (2002). Even Schneidewind and Hoffman, who differentiate their work from that of Endres on the basis that they are interested in ”programming” and not ”analysis and design” (p. 282), note the superiority of their error categories on the basis that some are designed to capture flawed ”mental processes” of the programmer in representing ideas within source code (1979, p.283).
3.1.3. Reflecting on the Research Model. In his paper Dependability: A Unifying Concept, Randell notes that clarifying the concepts underlying terminology related to dependability is difficult (Randell, 1998) (see also Appendix A for A Note About the Language of Error).

This is certainly true of the reports in this review. In part, this is an effect of their need to establish swiftly a conceptual basis upon which to present detailed empirical findings. In part, these differences in terminology reflect the passage of time. The studies reviewed span roughly twenty-seven years, and while they do not always explicitly relate their constructs to the other studies, or even explicitly define them, taken collectively, there is the sense that terms like interface evolved in this period from an anecdotal concept as reported by Endres to more precisely defined and measured constructs like those used by Perry and Evangelist.

Some of the researchers do admit to problems with coding reliability (Leszak et al., 2002), but suggest that their studies maintain a degree of internal reliability and thus produce valid results (Perry and Stieg, 1993). Nonetheless, it is difficult for readers to compare the findings, to pull from them a clear sense, for example, of what “design” is in software development, and how the findings related to design in the papers support or refute each other. This results in reports of refutation that actually report on different phenomena. And it results in fallacious claims of support. For example, Basili and Perricone (1984) suggest that like Endres, they found a majority of errors are “specification errors”, but their use of the term as a class of error is very different than Endres’ sense which sees specifications as one kind of problem in a much larger class of issues in the "area of understanding the problem, of problem communication, of the knowledge of possibilities and procedures for problem solving” (Endres, 1975, p. 331).

Another limitation of these reports related to terminology is that they don’t include examples of data to illustrate their classification choices. This makes it very difficult to understand the parameters of their analysis, how they determined membership in one category or exclusion from another. Even in cases in which description about the category is included, it often raises more questions than it answers. This is particularly true of the classification made by Perry and Evangelist, in their preliminary report on interface faults (1985). The study design does not include a formal component for validating categories with developers, a point noted as a limitation in the conclusions. The knowledge stated, therefore, is either drawn from the prior experience of the research team, or is reflected in some concrete way in the data. As reported, there is no way to evaluate the claims made.
This taxonomy includes a list of fifteen problem types, causes for the problems, and potential solutions. In examining this information, it becomes clear that this classification, like any, reflects the culture that produced it and cannot be considered to be objective or even neutral (Broughton, 2004) Under *Initialization/value errors* for example, the authors identify the cause as "Problems of this kinds are usually caused by simple oversight." How do they know this? Is there an example of an error that they classified in this way that demonstrates "simple oversight"? Under problem six, *Misuse of interface*, they note as the cause that the specifications were "probably not given with sufficient clarity". Similarly, under problem eleven, *Inadequate interface support*, the cause is given as "Design reviews are inadequate." The potential solution given states that problems in this category "frequently reflected the classical interface problem of one unit erroneously expecting another to conform to some standard – a bug that should be observed at the design stage". (Section 2.2 Analysis) Why should this be observed at the design stage? Even if every reader accepts this as fact, one is left wondering what is going on during development to allow such design flaws to so frequently persist.

### 3.1.4. Implications for Future Research.

Collectively, the root-cause analyses suggest two areas for future research. The first is that analytical perspective needs to be adjusted to include data about errors from the entire development cycle, and not just at the testing and integration stages. The data used in analysis should not be collected too long an interval of time after events have passed. Perry and Steig in particular note study design as a limitation, explaining that management at the organisation mandated that the study be voluntary, non-intrusive and anonymous, which resulted in the design of retrospective questioning techniques. Thus all underlying causes for errors were self reported, some period after the bugs were fixed, and the modification reports were closed (1993). This tallies with Endres’ view that root-cause analysis made upon results of testing and integration do not adequately represent the kinds of problems that crop up in earlier phases of development.

Surprisingly given their study’s experimental design, Schneidewind and Hoffman (1979) illustrate best the potential richness of this approach in the sample data they provide from their error analysis (see Appendix C). In this report, it is made clear that the root-cause analyst and programmer are the same person. The commentary included with the list of errors suggests many other possible avenues of investigation, as for example, when the programmer reports that he recognised the error while reading previously written code (errors 1-3), or that he was tired (error 28). What about reading previously written code caused the programmer to identify and address this error in particular?

---

1 This is never stated in the paper, but may be inferred from this data sample, and the citation to earlier Master’s Thesis work by the second author which appears to have formed the background to this study.
How did the two areas of the code relate? How did tiredness factor into the later error, what were the circumstances, and did this have other effects on the software?

The second is that studies should be made that examine the lower points of Endres’ human causes for errors taxonomy, those that fall into the "psychological" realm (Endres, 1975, p.331). Among those reviewed, only Endres’ study discusses in any detail the human aspects of errors, though humans do figure strongly in the findings of the others. Perry and Evangelist give several causes for their error categories related to human performance, include several mentions of inexperience (within their categories for example related to inadequate functionality and disagreements on functionality) (Perry and Evangelist, 1985, 1987). Leszak et al. (2002) report that it a mismatch between the technical skills required and those available among workers is the root cause of several errors. The report concluded that more human factors research is needed to investigate the inter-dependencies between the architectural and cultural elements of systems. Perry and Steig found that lack of information dominated underlying causes for errors and that knowledge intensive activities were suggested by participants as the most effective means of error prevention. They concluded that process should be altered to include "non-technological, people-intensive means of prevention" (Perry and Stieg, 1993). The studies led by Perry and Leszak conclude with suggestions for follow-up work using methods to investigate the human element of errors.

3.2. Coordination and Cooperation Studies. Unlike the root-cause analyses, these studies are not interested in the nature of errors and they do not follow a single model for design or analysis. Rather, they are interested in the environment in which software is created and in particular in the ways that tools and process support the coordination of activities required to create or maintain complex software systems. They also explore how work is mediated by the artefacts of software development: bug databases, code repositories, and in some cases, source code. As in the root-cause analyses, the studies primarily examine software development retrospectively, after code has been written or bug-fixes have been made, though Seaman and Basili (1997) and Ko et al. (2007) do include data collection of work as it is being done, and de Souza et al. (2004) and Aranda and Venolia (2009) do examine how work has been performed via interview, survey and documentary analysis.

These papers were selected for review because they use qualitative methods to examine software engineering practice, and their study designs are reported in enough detail and with enough clarity to suggest clear points of reference for the research proposed here. Seaman and Basili (1997) and de Souza et al. (2004) are notable because they examine activities other than bug-fixing. de Souza et al. (2004) provides a good model for exploring familiar software engineering themes like dependency in the context of
specific software features, in this case the application programming interface. Similarly Seaman and Basili (1997) demonstrate one way to study formal group interactions in software engineering. As proposed in this research, Ko et al. (2007) places the work of the individual developer at the centre of inquiry, examining problems of understanding, and personal strategies. The paper by Aranda and Venolia (2009) sounds a cautionary note. Though the authors initially intended to map the information flow of bug-fixing using Hutchins’ methodological framework for studying distributed cognition (Hutchins and Lintern, 1995), they discovered that the amount of data that emerged as related to individual bugs proved to be too rich and too complex to allow for comprehensive collection and collection. The authors opted to balance rich contextual detail with the ability to produce a replicable study and general results, a boundary which the research proposed here will also have to negotiate.

In merging the socially-oriented qualitative approaches of computer supported cooperative work (CSCW) with existing software analysis techniques, these studies provide new understandings of longstanding problems in software engineering. In so doing, they give alternative, rich insights into how developers perform their work, and into the ways artefacts reflect and shape that work. Given their focus on coordination and cooperation, they offer insights about factors that influence errors, but leave questions unanswered about their causes and natures. Like the root-cause analyses, they focus most intently on software-as-written, and aside from de Souza et al. (2004) examine most closely activities surrounding error detection and removal. Thus they offer a compelling new piece to the story of errors, and do so from a perspective that is firmly focused on the human aspects of software engineering. However they leave room for more focused examination of software as it is being written, and of the personal experience of developers.

In the remainder of this section, the studies mentioned above are profiled in more detail, with information given about study aims, design and findings.

3.2.1. An Empirical Study of Communication in Code Inspections. This study explored the nature of relationships among people engaged in code inspection processes and used this understanding to explain how those relationships impact the number of defects in software (Seaman and Basili, 1997). Methods used included documentary analysis, supplemented with observation and interview. The unit of analysis was the inspection meeting, meetings held at the studied organisation that included code authors, a moderator, code inspectors, and test inspectors. Data was collected from twenty-three inspection meetings held over a period of five months.

Findings suggested that meeting length correlates to the number of defects reported, number of defects does not correlate to program size, and included a number of hypotheses about factors influencing the number of defects reported during inspection. These
included hypotheses about the effects of interaction, collaboration, organisational structure, physical proximity, complexity and process on defects. It also included hypotheses about the effects of the same factors on the kinds of issues discussed in code inspection meetings.

These findings are compelling, suggesting support for Weinberg’s notion of the good effects of ego-less programming on the quality of software (Weinberg, 1998). This paper also benefits from rigorous design, using a mix of data collection methods and triangulation of data sources in analysis. The focus on code inspection meetings suggests that examining activities other than bug-fixing in software development may indeed yield evidence of things that go wrong.

3.2.2. Sometimes You Need to See Through Walls. This paper expanded the notion of dependency beyond its technical emphasis, demonstrating that the process of creating application programming interfaces is as much a human and organisational activity as it is an engineering technique (de Souza et al., 2004). This study examined one organisation’s use of application programming interfaces (APIs) to understand how they facilitate collaborative software development. Data was collected via an eleven week field study which included non-participant observation and semi-structured interviews. Documentary analysis included shared discussion databases, meeting invitations, product requests for software changes, and emails and instant messages exchanged among the software engineers.

The authors found that APIs facilitate coordination of activities in software development by serving as contracts among parties and by reifying organisational boundaries. They support collaboration by allowing teams of people to work independently. APIs also isolate developers, however, which results in less direct interaction between developers and in a lack of awareness both about what team members are doing and what they are capable of. The authors concluded with suggestions for improving collaborative tool development using social network analysis to allow developers to maintain awareness of what others are doing, even while they are working on parts of systems specifically designed to promote autonomy.

This study is notable because it examines an activity other than bug-fixing, and redefines a longstanding understanding of a qualitative attribute of software through the examination of a well-known feature of software architecture.

3.2.3. Information Needs in Collocated Software Development Teams. This paper includes a taxonomy of information needs that developers encounter while fixing bugs, starting from the premise that designing software with a consistent vision requires the consensus of many people (Ko et al., 2007). The authors argued that developers must have a detailed understanding of a system’s dependencies and behaviour, and suggested
that bugs can arise from the chasm between cause and symptom. Data was collected during a two-month period at Microsoft during ninety minute observation sessions. Seventeen developers participated. Questions that arose during development were cataloged, with key findings that co-workers are the most frequent source of information, suggesting a possible way to examine the effects of social interaction on mistakes.

Confirming the findings of the root-cause analyses, this paper also found that design problems and questions of original intent were the hardest to solve, themes that invite deeper examination. As proposed by this study, the authors suggest future work to probe more deeply into the work of other members of development teams, and to look more specifically at decisions made by individual developers.

3.2.4. The Secret Life of Bugs. Also at Microsoft, Aranda and Venolia made a case for developing rich bug histories in order to reveal the complex interdependencies of social, organisational and technical knowledge that influence and inform software maintenance (Aranda and Venolia, 2009). Starting with the thesis that electronic repositories as sufficient accounts of the history of bugs or work items requires validation, it argued that a description of the common coordination dynamics underlying bug histories is yet unwritten. Thus the goal of the paper was to “provide a rich, contextualised, work-item-centric account of coordination in bug fixing tasks” (p.299), and was achieved with a qualitative field study in two parts: a multiple-case exploratory case study of bug histories, and a survey of software professionals. The case study examined ten bugs, and interviewed twenty-six people to establish who was involved in the bug-fix and the contribution they made, the artefacts and tools used, the chronological timeline for information flow and coordination events, evidence related to each case, and a multi-dimensional history as represented in the bug database, other electronic sources, and a third representing all evidence, including data from interviews.

The authors found that the electronic records both within bug databases and in other electronic sources were fundamentally unreliable, erroneous or incomplete, and that the best way to fill in the gaps in these records was through direct interviewing of the participants involved in the bug-fixing activity, provided these people could be reached before too much time had passed. The authors found that bug records are lacking in six primary ways: they contain wrong or missing data, they do not provide an accurate record of the people involved in the bug, they do not record all relevant events, they do not capture information about group dynamics and their effect on bug-fixing efforts and they do not capture individual developers’ rationale for decisions taken while fixing bugs.
4. Research Approach

Previous studies have focused on the causes of errors and the environmental conditions surrounding bug fixing and maintenance. Studies span system types, sizes and domains, and rely almost without exception on analysis of bug reports filed during testing and integration, modification reports filed during software maintenance, or retrospectively administered interviews and questionnaires that probe for detail about these two activities.

Examination of mistakes that are made at other points during software development is not well addressed in the literature. One contribution of this research will thus be the application and evaluation of an appropriate methodological framework by which to properly address the research question. In addition to selecting appropriate methods for data collection, this framework must establish analytical focus. What constitutes a mistake worthy of examination? According to what terms does one characterise things that go wrong? Among what activities should one look? How should one relate understanding about things that go wrong to the considerable research on faults?

The root-cause analyses papers suggest two gaps with methodological implications. First, despite efforts to reduce concepts in software engineering into terminology that can be quantitatively measured, these studies demonstrate that the language used to describe errors is subtle and open to interpretation according to different environments, practices and orientations. Secondly, the methods used to-date for analyses do not permit nuanced understanding of the root causes of errors, and in particular of the role of human activity in error creation. This is due, in part to the fact that they rely on data collected only at certain points in the development process, after a mistake has been committed to software, caused something to go wrong, and been reported. Thus they cannot capture the genesis of faults, nor the range of other things that go wrong during software development but which are not caught. The coordination and cooperation studies support this view, showing that artefacts produced during bug reporting and fixing activities are incomplete, and demonstrating that qualitative data collection and analytical methods provide good means to fill in the gaps of understanding about these two activities.

For these reasons, this research will utilise qualitative methods to examine human activities in software development. The focus will be on understanding mistakes made more generally during software development and not just the consequences of mistakes as they manifest in faults. Curtis, Krasner and Iscoe’s *A Field Study of the Software Design Process for Large Systems* (1988) serves as an important model for this aim. The authors note that though software design is often spoken of as a “problem-solving” activity, development models don’t make use of existing empirical work on problem-solving.
in design (p.1269). By way of correction, the *layered behavioural model* developed in this study is orthogonal to the evolution of artefacts through developmental stages, examining instead factors that effect psychological, social and organisational processes. In this study, the approach yielded insights about recurrent problems related to design, including incomplete or uneven domain knowledge, fluctuating and conflicting requirements and communication and coordination breakdowns. The study also revealed that informal activity drives software design regardless of the formal processes that companies establish for making design decisions.

Other exemplars of qualitative approaches used in computing research that have been used to inform the work plan in the section that follows and which are expected to more formally guide study design, implementation and analysis are surveyed below.

Robinson, Segal and Sharp have reflected on the use of ethnographic techniques in empirical studies of software practice (2007), arguing that such approaches to data collection are necessary on the basis that the essential nature of work practice cannot be known *a priori*, and cannot be taken as “official”. The authors suggest that this kind of research is not hypothesis or theory driven, but instead is centred around the exploration of answers to open ended research questions. It is appropriately used to delineate problem spaces, to challenge received views, and to provide rich, narrative accounts of practice. Ethnographically-informed empirical work strives to be unobtrusive and to avoid control, intrusion, or the use of experiment. Rigour is achieved through triangulation of different data sources and via feedback with project participants. Analytical methods include archival research, discourse analysis and grounded theory, defined by the researchers as seeking disconfirmation and iterative development of understanding (get a page reference). Risks to the approach include potential bias on the part of researchers who are also software engineers, and thus members of the same culture, and ethical concerns as the approach often relies on informal and opportunistic data collection versus approaches that gain access to participants in a more formal, structured way. It has been used to study the adoption and evolution of software quality management systems, the emergence of object technology, professional end-user developers and agile software development. It draws upon qualitative research methods in education, anthropology and sociology.

For in-depth analysis of collaborative work, the tenets of Interaction Analysis as described by Jordan and Henderson (1995) may prove helpful. Interaction analysis holds that cognition is socially oriented and distributed, ”situated in the interactions among members of a particular community engaged with the material world” (p. 41). In practice, Interaction Analysis combines the use of ethnographically informed methods to establish contextual understanding of an environment with micro analytic techniques to
examine the details of interactions captured on video. It is necessarily interdisciplinary, drawing on fields such as socio-linguistics, ethnomethodology, conversation analysis, kinesics, proxemics, and ethology. The complete method presented by Jordan and Austin is intensive, involving iterative detailed study of video content by individual researchers, groups of researchers and with study participants.

In the study *Breakdowns and processes during the early activities of software design by professionals* Guindon et al. (1987) found that individual designers facing difficulties in high-level design activities exhibited three categories of breakdown: knowledge related breakdowns, such as a lack of specialised computing knowledge, experience or domain knowledge; cognitive limitations in the form of failures of short- or long-term memory, unreliable memory, and inadequate tool support for cognitive work, and; combined breakdowns characterised by aspects of knowledge and aspects of cognition. In the follow-up paper *Knowledge exploited by experts during software system design*, Guindon (1990) analysed the same design sessions for evidence of the specialised knowledge used by software designers when performing early design tasks. Her analysis included information about the kinds of new knowledge generated, the ways in which designers leverage existing knowledge, and included a set of heuristics used in seeking and selecting design solutions. Though these studies used protocol analysis in experimental studies, the examination of breakdowns remains an important model for studying the process of software design as it is unfolding, particularly as it may include “ineffective activities”. Similarly, Guindon’s particular articulation of the kinds and qualities of knowledge used in software design remain good foci for examining design sessions.

4.1. **Risks and Limitations.** Examining things that go wrong as a phenomenon distinct from faults is a novel approach, and is a potential risk to success. It may not be possible to find evidence of the phenomenon, or to sufficiently isolate it from the surrounding factors in a way that produces credible research results. This risk is mitigated by the large number of reflective accounts in the field that suggest quite strongly that the phenomenon exists. It is also mitigated by the proposed approach toward their examination using qualitative methods that include field studies. Though Curtis et al. note, quoting Weinberg’s *The Psychology of Programming* (1998), ”the idea of the programmer as a human being is not going to appeal to certain types of people” (1988, p. 1269; of Weinberg, 1971, p.279), field studies are known for being able to provide a much greater depth of understanding about human activities, and to produce results which are by some measures more valid because they are not artificially constructed (Babbie, 2004). The recent body of coordination and cooperation studies demonstrates the first of these points, and suggests, as do the findings of some of the root-cause papers, that the field of computing is keen to learn more about the programmer as a human being.
One limitation of the studies reviewed that this project will not be able to completely overcome is the need to rely on retrospective data. It likely will not be possible to have access to organisations at the moments when mistakes are being made, either individually or collectively. This fact will be mitigated by using interviewing techniques developed out of the critical incident method (Flanagan, 1954). In particular, the critical decision method as described by (Crandall et al., 2006) will be explored as a way to elicit focused information from developers about mistakes they are making. The critical decision method was developed to study individual decision making in naturalistic settings, with subsequent adaptations to examine group work, and everyday and critical incidents in the more distant past and in the "here-and-now" (Crandall et al., 2006, see for example, Chapter Five Incident-Based Cognitive Task Analysis: Helping Practitioners "Tell Stories"). As described, critical decision method requires a good working knowledge of cognition theory at the analysis stage, which the researcher will need to develop. Use of the method will also prohibit the researcher from maintaining an unobtrusive stance (Robinson et al., 2007), but it will facilitate the establishment of a balance between the collection of huge amounts of rich contextual data and the development of focused analyses (Aranda and Venolia, 2009) that can be applied to other software engineering research on faults.

Qualitative field studies pose other challenges. Organisational access, often a challenge in field studies (Crandall et al., 2006) can be even harder to attain when it will require reflection and sharing of information about mistakes (Perry, 2010). Even when it is possible to gain access, companies may place significant restrictions on data collection and reporting (Perry and Stieg, 1993) that may make the nuanced analysis suggested here difficult to achieve. Furthermore, in order to gather sufficient data to understand how mistakes evolve over time, development may need to be followed at key points over a longer period of time, or require access to a number of participants, requiring significant commitment on the part of participant organisations. Finally, field studies require that the researcher establish familiarity with the domains studied, in order to isolate, characterise and interpret data related to the domain as distinct from software engineering phenomena.

The computing literature supports these concerns. The papers studied report gaining access to organisations for significant periods of time (Basili and Perricone, 1984), include a period of scene setting data collection (Seaman and Basili, 1997) or self-limit their scope to focus on a single system in depth (Endres, 1975). Taken collectively, two reviewed coordination and cooperation papers (Ko et al., 2007; Aranda and Venolia, 2009) and several others conducted by Microsoft Research in recent years (Guo et al., 2011) demonstrate in-depth knowledge of the development culture at Microsoft.
To manage the risk of gaining sufficient organisational access in a timely manner and to facilitate the development of background and domain knowledge, this research proposes to make use of the researcher’s professional experience in academic computing. Access will be sought to organisations in the researcher’s professional network in humanities computing, and to departments within the Open University. This choice will have consequences. Field studies are known to be less reliable than other data collection methods, because the data collection is so personal, a weakness that can be exacerbated when the researcher is close to the environment studied. One way to improve reliability is to have more than one researcher collect and interpret data (Crandall et al., 2006), a solution that will not be possible in this research. Other ways to mitigate this problem which this research will employ are to collect data from multiple information sources, to seek disconfirmation of themes and findings via triangulation of sources during analysis and via follow-up data collection, and to carefully document the methodology followed so that other researchers can assess its credibility (Robinson et al., 2007).

To approximate longitudinal research, this project will include one study which depends primarily on documentary sources, either openly accessible in an open source project or via a comprehensive archive. Resources permitting, some retrospective interviews may be held with developers involved in the project.

4.2. **Contributions.** As Perry has noted, limiting data collection to a single system or domain provides deeper understanding at the expense of generalizability (2010). Aranda and Venolia (2009) found that their original aim to describe a general lifecycle for bugs and bug fixing proved to be impossible as the histories they uncovered were too “rich, varied, and context dependent” (p.304). This research does not propose to generalise about the mistakes made in all of software engineering. Findings may not even generalise to the kinds of software engineering practiced in the studies reviewed, that is to say in large scale industrial development.

The contributions of this research will not include a set of general guidelines or hypotheses for improving software engineering practice. Instead they will comprise:

- a clear articulation of the problem space of things that go wrong during software development;
- a detailed account of qualitative methods that can be effectively applied to examining this problem space; and
- a culturally defined and contextually meaningful model of such mistakes from which hypotheses may be generated for future examination in other contexts.

In this preliminary study an analytical framework was developed to examine difficulties encountered in early collaborative design discussion\(^2\). The framework drew upon principles of design cognition as defined by Nigel Cross and the "kinds of knowledge" identified by Raymonde Guindon and colleagues. This study posited that at least some "faulty ideas" or design flaws begin as difficulties that emerge in discussion: disagreements, mis-communications, or "wrong turns" taken by designers engaged in collaborative effort to identify and structure design problems, and to formulate appropriate solutions. To explore these positions, the study asked: How are difficulties encountered and overcome in early design work? What can detailed study of difficult episodes in early design discussion tell us about how to study flaws as they develop within later software design activities?

In answer, this study established an analytical framework for studying flaws that originate during design discussions. Analysis began with the creation of a rich transcription of one design session from the NSF funded International workshop "Studying Professional Software Design" (SPSD), held February 8th-10th, 2010, at the University of California, Irvine. The goal of this workshop was to collect observations and insights into software design, drawing on theories and methods from a variety of research disciplines including software engineering, design studies, human-computer interaction, cognitive science and psychology. Workshop participants analysed a common set of data comprised of videos and transcripts of three pairs of professional software designers. More information about the workshop may be found at: http://www.ics.uci.edu/design-workshop/.

The rich transcription included information about gesture and whiteboard work, and captured additional linguistic content. This was followed by segmentation of the transcription to isolate particular events for study; each event was additionally broken down into episodes: distinct periods within the session in which the event was discussed. The rich transcription corresponding to individual episodes was converted into a columnar catalog. The catalog included cross-referenced information about gestures used; whiteboard work, specifically sketching or amending existing sketches, and; references made to the design prompt, or periods of examination or re-examination of the design prompt. Within the broader framework of Cross’ principles of design cognition, individual exchanges were examined for evidence of the particular kinds of knowledge exploited by designers as identified by Guindon\(^3\).

5.1. Reflection. Though effective, the method used in this study was extremely time consuming. In future work, the use of video and the development of rich transcriptions

\(^2\)The full report may be downloaded at: http://users.mct.open.ac.uk/tl2768/pilot-SPSD/

\(^3\)The catalog used in analysis can be viewed in Appendix D: Guindon’s Kinds of Knowledge.
may be warranted, but elements of gesture, whiteboard and prompt activity only need to be added to episode analysis. The analytical framework is promising, but should be validated through examination of additional laboratory design sessions and via confirmation of findings with participants. In addition, the framework should be tested on early design activities occurring in practice, and on other design discussions occurring in later stages of development, including periods of software integration, bug-fixing or in the specification and development of new features.

6. Work plan

6.1. Overview. This section describes the work plan for researching mistakes. In total, three studies are proposed: two to be completed in Year Two, and a third to be completed in Year Three.

<table>
<thead>
<tr>
<th>Year</th>
<th>Calendar Dates</th>
<th>Summary of Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Feb 2010-Oct 2010</td>
<td>Proposal development</td>
</tr>
<tr>
<td>1</td>
<td>Oct 2011-Jan 2012</td>
<td>Probation viva; refine data collection and analysis plans for Year Two</td>
</tr>
<tr>
<td>2</td>
<td>Feb 2012 - Jan 2013</td>
<td>Studies 1-2; dissertation draft</td>
</tr>
<tr>
<td>3</td>
<td>Feb 2013 - Jan 2014</td>
<td>Study 3; dissertation draft</td>
</tr>
</tbody>
</table>

The number of weeks per year is calculated based on the assumption that the full amount of allotted holiday time will be taken, and that four weeks contingency time will be required (i.e. 40 weeks/year = 52 weeks - 8 weeks holiday - 4 weeks contingency)

6.1.1. Contingencies. Three major contingencies must be considered.

The research presented in Years Two and Three will be undertaken following a year long period of maternity leave. In order to facilitate adjustment at a later time, timeframes for tasks are given in terms of periods in weeks, but are not assigned specific calendar dates.

In addition, this project depends to a great extent on negotiating sufficient organisational access. At the time of writing this proposal, contact has been made with several organisations, however given the period of maternity leave, no firm plans for site visits have been made.

The aim for all of the studies is to replace the research and guiding questions presented in this report with questions about mistakes that emerge from the field, and to collect sufficient data to answer them (Meloy, 2002). To this end, snowball and convenience sampling will be used to recruit participants (Babbie, 2004), the exact number of which cannot be known beforehand. In order to develop a preliminary schedule, this work plan assumes the following for the studies in Year Two: Attempts will be made to have access
to the same kinds and amounts of data from at least five organisations for each study, and to collect data from between 20 and 25 individual participants.

6.2. **Year One.** The period of Year One subsequent to passage of the probation mini-viva will be used to refine data collection and analysis plans for Year Two, and to finalise negotiations and schedules for access to organisations.

6.3. **Year Two.** Two field studies will examine aspects of things that go wrong as they relate to different behavioural levels (Curtis et al., 1988). Data about mistakes will be collected, analysed and reported in two separate studies that take different units of analysis as their focus, namely the individual, and the team. Developers will be recruited via convenience and snowball sampling for retrospective interviews using techniques drawn from methods associated with cognitive engineering (Crandall et al., 2006). These data collection methods may be supplemented with periods of *in situ* observation to facilitate understanding about how selected environments operate and to enrich understanding about how people work in these environments.

Data collection will be made during short, intensive periods of between five and ten business days, with some followup data to be collected at other times. On-site data collection at each organisation will be preceded by scene setting interviews, during which organisational and project materials will be collected for documentary analysis (Robinson et al., 2007; Crandall et al., 2006). Documents to be examined may include such items as software code; communications related to design and programming decisions, and; specification, requirement and delivery documentation. In addition to detailed catalogs of empirical data about the mistakes reported, these studies will produce a culturally meaningful (Meloy, 2002) description or model of how mistakes develop in the environments studied.

Organisational visits are presented sequentially in the text that follows, but it is quite likely that visits to different places will occur concurrently. Analysis will likewise be ongoing and iterative throughout the year.

6.3.1. **Study One: Working Alone.** This study will explore the ways in which individual developers identify, diagnose, and correct mistakes. The aim is to understand personal strategies and techniques for dealing with mistakes, as well as to identify how individuals reach out for support in the process: for example via other employees, and external and internal resources. Activities examined will include day-to-day development work, including programming, design and analysis tasks, documentation and reporting.

Data will primarily be collected via retrospective interview though short periods of observation may be made of the developer as opportunities arise. The study will also
Table 1. Year Two Tasks and Timeframes (40 weeks = 52 weeks - 8 weeks holiday - 4 weeks contingency)

<table>
<thead>
<tr>
<th>Timeframe</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 weeks</td>
<td>Organisation 1 data collection</td>
</tr>
<tr>
<td>2 weeks</td>
<td>Organisation 2 data collection</td>
</tr>
<tr>
<td>2 weeks</td>
<td>Organisation 3 data collection</td>
</tr>
<tr>
<td>2 weeks</td>
<td>Organisation 4 data collection</td>
</tr>
<tr>
<td>2 weeks</td>
<td>Organisation 5 data collection/prior organisation follow-up (contingency; as required)</td>
</tr>
<tr>
<td>2 weeks</td>
<td>Organisation 6 data collection/prior organisation follow-up (contingency; as required)</td>
</tr>
<tr>
<td>2 weeks</td>
<td>Organisation 7 data collection/prior organisation follow-up (contingency; as required)</td>
</tr>
<tr>
<td>2 weeks</td>
<td>Organisation 8 data collection/prior organisation follow-up (contingency; as required)</td>
</tr>
<tr>
<td>12 weeks</td>
<td>Study 1 analysis and reporting</td>
</tr>
<tr>
<td>12 weeks</td>
<td>Study 2 analysis and reporting</td>
</tr>
</tbody>
</table>

include the selection of a subset of developers from each organisation for follow-up contact. This contact may be used to verify understanding about events initially discussed, and may include collection of diary or interview data to track what happened next with targeted issues, and how individuals’ thinking developed about key issues subsequent to the initial interview.

Possible data sources for triangulation:

- Observation, field notes
- Semi-structured interview, field notes
- Documentation (personal notes, reporting, design or other descriptions)
- Individual background
- Project context
- Organisational context

6.3.2. Study Two: Working with Others. Drawing upon the preliminary study, this study will examine the ways in which group dynamics influence and shape mistakes. The aim is to understand collective strategies and techniques for dealing with mistakes, as well as to understand how groups support this process: for example via discussion, the use of descriptions, and institutional knowledge. Activities examined will include early design sessions, ongoing design and architectural meetings (both formal and informal), and code inspection meetings.
Table 2. Year Three Tasks and Timeframes (40 weeks = 52 weeks - 8 weeks holiday - 4 weeks contingency)

<table>
<thead>
<tr>
<th>Timeframe</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 weeks</td>
<td>Cross-study analysis; Study 3 refine research questions, aims and methods</td>
</tr>
<tr>
<td>12 weeks</td>
<td>Study 3 Data collection</td>
</tr>
<tr>
<td>12 weeks</td>
<td>Dissertation draft</td>
</tr>
</tbody>
</table>

Data will be collected via retrospective interview coupled with observation of both formal and informal group activities. The study may also include the selection of a subset of participants for follow-up contact. This contact may be used to verify understanding about initial sessions, and may entail additional interviewing to track what happened next with targeted issues.

Possible data sources for triangulation:

- Observation, field notes
- Video-taped sessions
- Semi-structured interviews
- Documentation (design or other descriptions; communications)
- Team context
- Project context
- Organisational context

6.4. **Year Three.** This year will be devoted to conducting one study and to dissertation drafting. It will commence with a period of comparative analysis of the completed studies, which will include refinement of research questions, aims and methods. Cross-study analysis should be completed before Study 3 data collection and analysis occur and before the dissertation draft can be completed. However some concurrent work may be possible. For example project selection and background information collection can commence before cross-study analysis is complete.

6.4.1. **Study Three: Project Work.** This study will examine mistakes as they are situated within project contexts. Sub-units of analysis will be the descriptions produced in the life of a project: diagrams, specifications, process documents, source code, and communications in the form of modification and bug reports. The identification of mistakes in these descriptions will be colligated (Anderson, 1997) with the model of mistakes that emerged from the individual and group levels, i.e. links will be drawn where possible between mistakes made in individual and group work and their subsequent representation at the description level. This study has one aim: to develop the emerging model to
account for the trajectory of a mistake from the psychological and behavioural realms back to the software record.

This study will depend primarily on documentary sources, either openly accessible in an open source project or held within an organisational or institutional archive. Resources permitting, some retrospective interviews may be held with developers involved in the project.

Possible data sources for triangulation:

• Specifications
• Design Diagrams
• Source code
• Bug and Modification reports
• Communications
• Team context
• Project context
• Organisational Context

6.4.2. Dissertation Draft. Writing of the dissertation is expected to begin with incremental reporting of individual studies and ongoing review and synthesis of literature. Year Three will conclude with the submission of a dissertation draft to the supervisory committee. It is expected that the final submission and defence of the dissertation will be undertaken after the period of funding ends, in Year Four (February 2014 - January 2015).

7. Conclusion

Software engineering research will benefit from detailed examination of mistakes made while software is being developed - the study of those things that go wrong during initial design and development and within cycles of development, and which either fade away, are caught and fixed or ignored, or which may go on to lurk within the software as latent faults or to activate and threaten operational failure (Avižienis et al., 2004). Furthermore, software engineering needs to examine mistakes as a product and consequence of human effort, to probe more deeply into the gap between original intent and outcome (Endres, 1975), to understand why developers report so many errors as having causes related to problem understanding, problem definition and communication. In answer, this research will examine such mistakes as they come into being, to capture how they are made individually and among software developers, to recode their character, how they are found, and how they are dealt with.

The outcomes of this research will include a large body of empirical evidence of mistakes, and a detailed account of methods used in their examination. Analysis of this
data will produce a model that links behavioural and process oriented aspects of software development.

This research aligns itself most closely with other research into the cooperative and human aspects of software engineering. It may also prove useful to branches of software engineering research concerned with software dependability.
Appendix A. A Note About the Language of Error

In his paper *Dependability: A Unifying Concept*, Randell noted that clarifying the concepts underlying terminology related to failure is difficult (Randell, 1998). System boundaries are fluid, systems and the artefacts used to represent them can be complex, judgements about what causes failure may be subjective and subject to perception and attitudes, and the very mechanisms designed to prevent failures are themselves failure-prone. The complexity of the topic has resulted in the reinvention and renaming of related concepts by different communities, and a tendency to overlook the ways in which different means of achieving dependability (and thus preventing failure) are relevant to one another.

As will be shown, this is true of the pieces reviewed in this proposal, and also hinders efforts to frame the area of investigation undertaken by this project. Distinguishing analytical terms describing concepts related to error from the language used within the literature is also difficult because the thesis presented here is emergent. It argues that any such analytical terms should be derived from an analysis of data collected from field work. At the present moment, it is thus dependent upon sources that exhibit the factors identified by Randell, and possess historic dimensionality.

Nevertheless, some terms must be agreed to for the sake of advancing the work.

Randell’s subsequent taxonomic work with Avižienis and Laprie identified dependability in both qualitative and quantitative terms (Avižienis et al., 2004, p. 5). Qualitatively, it is the ability to deliver service that can be justifiably trusted, with an emphasis on the need for justification of trust. Quantitatively, dependability is the ability to avoid service failures that are more frequent and more severe than is acceptable to the user. The authors of this paper note that over several decades, software engineering research has developed the notion of dependability as an overarching concept that subsumes reliability and other software engineering attributes like availability, safety, confidentiality, integrity and maintainability.

In this research proposal, the qualitative sense of dependability is employed to describe in general terms the aims and efforts of software engineering research that is itself concerned with the quantitative sense of dependability, and to identify commentary in the broader computing discourse that qualitatively reflects on what dependable software is.

In the same paper, Avižienis et al. define errors as deviations in an external system state from the correct service state (Avižienis et al., 2004, p. 4). The determined or hypothesised cause of an error is referred to as a fault. A fault is active when it causes an error, and is otherwise dormant. Avižienis et al. leave unstated that faults so defined are mediated, that is, they are represented within software artefacts. This connotation
is better understood from the definition of error given in the model root-cause analysis reviewed within the literature review (Endres, 1975). In that paper, Endres uses the term *error* in terms analogous to the Avažienis et al. concept of *fault*. However Endres additionally demarcates the term by indicating that the actual error equates to the correction made - that is to say that errors are mediated, they are written into software and can be located and removed. He acknowledges the existence of, but rejects for the purpose of his research, real errors which may lie too deeply within the source code, or which may present too great a risk to properly fix. Furthermore he equates number of errors with number of problems reported in the test cases, thereby omitting the fact that fixing problems may require removing more than one error from the source code. Unlike Avažienis et al., Endres does not use the term correct but rather uses terms like intended and original intent to describe the gap between what is expected to happen in a piece of software and what actually does.

In this proposal, the distinction made by Avažienis et al. between errors and faults is considered to be too fine-grained, and the term correct is felt to be similarly overly precise. Instead, their term fault, Endres’ term error and a third term widely used in the software engineering community – bug – are used interchangeably to describe elements of software as written which may produce deviation in the external system state from that which was expected.

*Errors, faults and bugs* so defined are not the phenomena proposed for examination by this research. This research is concerned with the other causes for errors given by Endres, and in particular with those he identifies as related to the fact that programming is a ”human activity” (Endres, 1975, p. 331). Elsewhere in the paper, he refers to the things that go wrong and the personal theories and experience that programmers develop over time as a result of dealing with them (p. 327). Similarly, this research is interested in capturing the detail of the things that go wrong while developers are making software.

To understand this perspective, the nomenclature provided in *Going Amiss in Experimental Research* (Hon et al., 2009) is helpful. This text examines the notion of going amiss in experimental research from an historical and philosophical perspective. It is notable in that it encapsulates concepts of error and mistake, but also considers the many other ”kinds of pitfalls” (p.3) that scientists encounter in their work. Furthermore, it distinguishes between terms applied by scientists when speaking retrospectively about a problem encountered, and those which are used by scientists to describe things that are going amiss in the present moment. The former category includes terms like ill-formed questions, misguided expectations, misinterpretations, anomalies, discordance, inconsistencies, while the latter includes notions like surprise, puzzlement and confusion and
acknowledges that sometimes scientists reach a dead end – an inability to understand what has gone wrong or why.

This research situates itself at the intersection of the notions of Endres and Hon et al.. It examines the things that go wrong while making software. It takes into account the understanding and theories that developers form in the process, the strategies they use to manage problems encountered, and the ways they communicate about the experience. It acknowledges that even when software succeeds and everything ”turns out right” (Hon et al., 2009, p.2), the path to that success includes ”pitfalls and confusions” (ibid.), errors as they are commonly understood and examined in software engineering, but also ”misguided conceptions, dead ends, and reorientations.” (ibid.).

In the main, therefore, this proposal will refer to the phenomenon under investigation as the things that go wrong. Where a more precise term is required, things that go wrong shall be referred to as mistakes. While it is understood that mistake in its noun form connotes the error made (and therefore is mediated in some way within an object or artefact), here it is used as defined in verb form by the Merriam Webster Dictionary. So taken, it means ”to blunder in the choice of”, ”to misunderstand the meaning of”, or ”to identify wrongly: confuse with another” (Webster, 2011). This usage tallies with James Reason’s appropriation of the term to mean errors that arise ”in the planning and thinking process” (Sasou and Reason, 1999, p. 1).
APPENDIX B. PERSPECTIVES ON FAILURE ANALYSIS IN COMPUTING

This section includes a brief survey of analyses treating complete operational failure, followed by a similar section on analyses that treat smaller aspects of failure within systems. It makes no claim of comprehensiveness, but does assert that the pieces included are representative of the kinds of studies identified. It was written in March 2010 and formed the synthesis of the initial literature review performed for this research proposal. A portion of this was also presented at the Open University CRC post-graduate conference in the summer of 2010.

Selection of articles was made by performing detailed keyword searching of journals, chasing citations within articles, reviews\(^4\), position papers and experiments, and a search of journalistic sources and software engineering course work and syllabi that specifically address software failure. Materials were selected from work dating back to the 1960’s with some representation from the ’80s and ’90s; the majority of materials are from the first decade of the 21st century. Out of approximately eighty items surveyed, roughly a third of the materials came from books, technical reports or conference proceedings; the remainder are journal articles.

The journals which were systematically reviewed (including articles up to the end of February 2010) included:

- Software Practice and Experience
- Software Process: Improvement and Practice
- Software Testing, Verification and Reliability

Other software engineering and computer science journals which have significant representation in the survey (i.e. more than one source) include:

- Communications of the ACM
- IEEE Computer
- IEEE Software
- IEEE Transactions on Software Engineering

B.1. System Analyses. System failure analysis examines *systems-of-systems*(Randell, 1998, 2007) to identify weak elements in complex organisational, operational and software systems. As in other branches of engineering\(^5\), these analyses are retrospective, performed after a service outage as a way to understand what went wrong, and who was responsible. In general, these analyses examine sudden and progressive failures, however this should be treated as a soft categorisation. Systems which primarily exhibit

\(^4\)Reviews here characterised to include roadmaps, state- of the art and -of the discipline pieces

\(^5\)See, for example (Levy et al., 2002)
characteristics of progressive failure could suddenly fail, and sudden failures may show evidence of progressive issues when analysed.

Sudden failure is service outage on a large scale, often involving critical piece of software. Individual or multiple faults become active at a moment in time or within a clearly bounded interval of time, and result in a large, catastrophic or spectacular system failure (Leveson and Turner, 1993; Nuseibeh, 1997; Thein Than et al., 2009).

Progressive Failure arises in software systems that are deemed “good enough” to be released into production but which include significant problems that require maintenance, redesign and redevelopment, or that result in overextended resource allocation. Often this software is conceived and implemented within an already failing or flawed organisational or system initiative (Ince, 2010; Randell, 2007; Dalcher and Tully, 2002).

The case studies produced by these analyses often do not conclude with specific, precise reasons for failure, instead offering identifications of the system or subsystem that failed, and general recommendations for improvement going forward. Even when they do isolate weaknesses in the processes of software creation or in particular software components, they do not tend to produce general frameworks or models that can be extended to improve software engineering practice. One notable exception to this is the 2009 paper *Are Your Lights Off? Using Problem Frames to Diagnose System Failures* (Thein Than et al., 2009), an analysis that uses a specific framework to retrospectively analyse one aspect of the development process in a system that suffered sudden failure.

These treatments are found within software engineering literature, but are also produced within “grey” literature (Easterbrook, 2005; Dix, 2003), that is, in unpublished workshop and conference presentations, and course work materials, and in journalistic accounts of failure (Charrette, 2005; Barker, 2007; Garfinkel, 2005; Bogdanich, 2010). In the latter case, accounts can include strong undercurrents of anti-utopianism, though unlike the social analyses characterized by Kling (1994), these are past rather than future looking. In other aspects however, they conform to the genre he identified: they universalise technological experience, can take extreme value positions, and describe technology as a dominant force in social interactions. Unlike popular accounts of catastrophic failure, workshop and conference presentations tend not to be so negative or to engender a sense of despair. However, they, like the former, present their cases simply and draw on spectacular examples, making them compelling and easy to understand, characteristics that Kling would suggest also make them influential in shaping both discussion about computing, and the directions that computing research takes.

B.2. Dependability Analyses. Dependability analyses of treat smaller aspects or attributes of software engineering as they contribute to or hinder the broad goal of creating dependable software. Thus they are not concerned with failure *per se*, but rather in how
how to make software better so as to avoid it. They are found in studies that develop or test existing methods and techniques to provide dependable services and may be categorised into four kinds: fault prevention, fault tolerance, fault removal, and fault forecasting (Randell, 1998).

As their name suggests, fault prevention studies (Shaw, 2002; Thein Than et al., 2009) work to prevent the introduction of faults during design and development. Requirements engineering, structured design and programming methods, formal methods and software reuse are areas of software engineering research that might be placed into this category.

Fault removal studies (Hanebutte and Oman, 2005; Butler et al., 2010; Zou, 2003; Pugh, 2009; Briand et al., 2003; Cataldo et al., 2009) seek ways to remove faults that are written into software during the software verification and validation phases of the development lifecycle. Areas of software engineering that work to promote dependability using this mean are software testing, formal inspection, and formal design proofs. As with prevention, removal techniques cannot ensure that all faults are removed from a system, because they can only determine whether or not software matches the specified required behaviour; they cannot determine that something was left unspecified.

Fault forecasting (Bertolino and Strigini, 1998), as with fault removal, is employed during validation of software to indicate the presence of faults and to predict the occurrence of operational failures. It can be used to determine whether additional testing or other means should be applied to software before it is released.

Fault tolerance techniques (Sözer et al., 2009) enable systems to tolerate faults that are not removed prior to release. They do this by allowing operations to degrade gracefully and to recover from errors with an aim toward preventing complete operational failure. Fault tolerance is formed of a larger set of processes that include error detection, diagnosis, containment and recovery.

Means analyses are empirical and employ a range of methods including statistics, program analysis, case study, formal-methods and systems analysis. Often a study will examine a single part of the development process, such as requirements engineering with a corresponding focus on a single dependability mean, though there are exceptions to this pattern. See for example (Magalhes et al., 2009) which develops a methodology for developing debuggable software that includes efforts to achieve each of the four means. In practice, the studies are either experimental and therefore not temporally bound or retrospective in the sense that they test techniques on existing bodies of software. All of the studies, however, can be considered to be prospective, in that they are working toward the aim of improving process and practice in future development initiatives.

\(^{6}\)Definitions of means and synthesis of software engineering areas that employ each are derived from (Pullum, 2001).
The studies reviewed to-date do include “grey literature” (Pugh, 2009) and reference works (Pullum, 2001), but no evidence of journalistic accounts has been found. This is likely simply a limitation of the current literature search; it is expected that technical websites and other publications like ACM Communications will include a good deal of commentary about these kinds of studies.
Appendix C. Endres’ Classification

This following taxonomy of error distribution by type of error was the primary outcome of Endres’ analysis into the root-causes of software errors. This taxonomy included three main groups: one to categorise errors related to problem understanding (Group A in Appendix A below), one related to implementation (Group B) and one related to mechanical errors such as spelling, or errors in integrating modules (Group C). The taxonomy included a breakdown of factors contributing to the errors for several subgroups in classes A and B, and offered descriptive statistics for future detection of errors in classes A and B.

Subsequent studies using the research model established by Endres produced taxonomies of varying detail and composition. In all cases, they form the centrepiece of analysis. By examining individual categories and labels and the discourse that surrounds them, it is possible to glean something of the cultures that produced them, and of the purposes for which they were created. To read more about this, see the section Reflecting on the Research Model in the literature review, above.
Figure 1. Endres (1975)

<table>
<thead>
<tr>
<th>Code origin of module</th>
<th>Total number of modules with errors</th>
<th>Number of modules with errors</th>
<th>Percentage of modules with errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>New only</td>
<td>169</td>
<td>81</td>
<td>48</td>
</tr>
<tr>
<td>Old + new</td>
<td>352</td>
<td>271</td>
<td>48</td>
</tr>
<tr>
<td>Total</td>
<td>521</td>
<td>352</td>
<td>68</td>
</tr>
</tbody>
</table>

| New only              | 169                                | 254                         | 1.5                             |
| Old + new             | 352                                | 271                         | 1.3                             |
| Total                 | 521                                | 525                         | 1.6                             |

<table>
<thead>
<tr>
<th>Size of new code</th>
<th>Number of errors</th>
<th>Errors per module</th>
<th>% of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>New only</td>
<td>153</td>
<td>254</td>
<td>4.8</td>
</tr>
<tr>
<td>Old + new</td>
<td>352</td>
<td>271</td>
<td>2.7</td>
</tr>
<tr>
<td>Total</td>
<td>505</td>
<td>525</td>
<td>4.9</td>
</tr>
</tbody>
</table>

Figure 5. Frequency of errors

A1 Machine Configuration and Architecture
(a) Type of device or device feature not considered. Valid I/O command taken for invalid
(b) I/O error condition or device status handled incorrectly or not at all
(c) I/O command used incorrectly, or simulated incorrectly (i.e., simulating one device by another one)
(d) Error statistics for a device not generated or generated needlessly
(a, c) External operation mode of a device handled incorrectly

A2 Dynamic Behaviour and Communication between Processes
(a) System state which was entered dynamically not identified exactly
(b) In case of sequential transition to another program, the new program was not started up until the first process did not find the expected parameters (e.g., register contents)
(c) Registers and control blocks used repeatedly were not saved. Intersect destroys information which is still needed
(d) Interrupts were enabled which could have been masked out. Other interrupts were masked out and thus ignored, although they were important for the functioning of the system
(e) Processes get new data (e.g., opening a file) while reading, wrong sequence, wrong return brand
(f) Incorrect resource allocation, deadlock, allocation of non-existing or of previously assigned resources
(g) If a new function was not generated (or not activated) a related subsequent function was not eliminated at system generation time (or not switched off at run time)

Figure 6-1. Types of errors - Group A

A1 Functions Offered
(a) Functions are completed, no necessary for the intended use of the system
(b) Functions are added, or generalized, although not originally intended
(c) Functions are completed in order to handle extreme cases and other exceptional situations
(d) Functions are changed in order to improve usability, security, compatibility etc.
(e) Functions are added externally (e.g., product strategy)
(f) Defaults for limited parameters changed
(g) Message for operator/user added
(h) Function is eliminated, because no longer needed

Figure 6-2. Types of Errors-Group A1

A2 Addressability (in the sense of the assembler)
(a) Reference to names
(b) Counting and calculating
(c) Mask and comparisons
(d) Estimation of range limits (for addresses and parameters)
(e) Placing of instructions within a module, bad lines

Figure 6-5. Types of errors-Group B

Figure 1. Endres (1975)
Figure 2. Endres (1975)
Figure 3. Endres (1975)

**Figure 7.1. Error factors—Group A.**

- **A1** Functions Offered
  1. Quality of specifications
  2. Experience with similar systems
  3. Statistical information on user profiles, data volumes, operating modes
  4. Clarification of and concentration on worst cases, exceptional situations
  5. Self-discipline with one's own "bright-ideas" and external suggestions

**Figure 7.2. Error factors—Group B.**

- **B2** Addressability
  1. Extension of symbolic addressing
  2. Extendibility of address space
  3. Estimation of address space for each routine ("need to know")

**Figure 7.3. Error factors—Group C.**

- **C4** Counting and Calculating
  1. Self-programming data and areas
  2. Powerful loop commands to be linked to data descriptions (e.g., retrieve for all entries of a label)
  3. Tables or easily-accessible transformation routines for the calculation of disk addresses or track capacities
  4. More regularity in the addressing structure for all devices: symbolic addressing

**Figure 7.4. Error factors—Group D.**

- **D1** Initialization
  1. Forced initialization or warning message by language translator if initialization is missing
  2. Automatic adaptation of operations to field length (e.g., clear the whole field)
  3. Analysis of routines for effect on control blocks, registers, and data fields
  4. Specification of explicit and implicit parameters of allowed and expected value ranges

**Figure 7.5. Error factors—Group E.**

- **E3** Reference to Names
  1. Syntax of names
  2. Possibility of qualification for names
  3. Representation of the role of a field and indication of routines with access rights
  4. Admissible addressing of tables

**Figure 7.6. Error factors—Group F.**

- **F4** Detection Method

<table>
<thead>
<tr>
<th>Type of Error</th>
<th>Examination of spec by others</th>
<th>Format description needed</th>
<th>Simulation, building</th>
<th>Program inspection by others</th>
<th>Test runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 Max/min configuration + priorities</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>A2 Dynamic behaviour + communication</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>C1 Comments</td>
<td>10</td>
<td>10</td>
<td>-</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>A3 Output listings + formats</td>
<td>10</td>
<td>10</td>
<td>-</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>A4 Performance</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

**Figure 7.7. Error detection—Group A.**

<table>
<thead>
<tr>
<th>Type of Error</th>
<th>Program inspection by others</th>
<th>Floyd/Hoare method of proof</th>
<th>Simulation of test runs</th>
<th>Test runs by the programmer</th>
<th>Test runs by others</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1 Initialization</td>
<td>10</td>
<td>10</td>
<td>-</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>B2 Addressability</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>B3 Reference to names</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>B4 Counting and calculating</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>B5 Masks and comparisons</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>B6 Estimation of range limits</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

**Figure 7.8. Error detection—Group B.**

---

Figure 3. Endres (1975)
Appendix D. Root-Cause Analyses Profile

The research model established by Endres was subsequently adopted by a number of other software engineering researchers. The “bones” of these studies are presented below, with information given about individual study aims, characteristics of study design, and the environment under investigation. To read a comparative examination of other details of these studies, see the section Following the Research Model in the literature review, above.


<table>
<thead>
<tr>
<th>Hypotheses/Aims</th>
<th>Data Characteristics</th>
<th>Software Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypothesis: Program structure has a significant effect on error making, detection, and correction. Aim: To find a complexity measure that can be used to guide program design and resource allocation in debugging and testing.</td>
<td>173 errors, 64 errors deemed to be potentially relevant to complexity of structure; purpose-built code.</td>
<td>Four projects were programmed by the same programmer in Algol W for execution on the IBM360/67; n/a.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Hypotheses/Aims</th>
<th>Data Characteristics</th>
<th>Software Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aim: To analyze the relationships between environmental factors and errors reported during software development and maintenance.</td>
<td>231 change report forms, created by programmers over a period of 33 months. Reports were verified by team manager, validated by research team; new development, but existing code re-purposed in some cases</td>
<td>Approximately 90,000 lines of code primarily written in Fortran for execution on an IBM 360; aerospace (satellite planning studies).</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Hypotheses/Aims</th>
<th>Data Characteristics</th>
<th>Software Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypothesis: Interfaces are a source of problems in the development and evolution of large system software.</td>
<td>94 randomly selected modification reports submitted by testers that satisfied the operational definition of interface; 85 contained sufficient data for the study; software evolution.</td>
<td>350,000 non-commentary source lines written in the C programming language; fault reports written against global header files. Domain unreported, researchers affiliated with Bell Labs and MCC.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Hypotheses/Aims</th>
<th>Data Characteristics</th>
<th>Software Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aim: To determine general and application specific encountered during software evolution. Aim: To determine problems are found. Aim: To determine when problems are found.</td>
<td>Total sample size unreported; 68% of surveys were returned in each of two surveys; software evolution.</td>
<td>1,000,000 non-commentary source code lines, distributed real-time system written in C on UNIX; telecommunications (AT&amp;T).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hypotheses/Aims</th>
<th>Data Characteristics</th>
<th>Software Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aim: To analyze defect modification reports; establish root causes. Aim: To analyze customer-reported modification reports Aim: To propose improvement actions to reduce critical defects and to lower rework cost</td>
<td>427 Modification Reports representing 13 domains (functional units of software); new development and evolution.</td>
<td>900,000 non-commentary source code lines, 51% of which is newly developed software. Language and environment unreported; telecommunications (Lucent).</td>
</tr>
</tbody>
</table>
Appendix E. Schneidewind’s and Hoffman’s Error Listing with Comments

The commentary included with the list of errors in this 1979 study by Schneidewind and Hoffman suggests many other possible avenues of investigation that might be pursued in an effort to understand the causes of errors. For example, when the programmer reports that he recognised the error while reading previously written code (errors 1-3), or that he was tired (error 28) one begins to grasp the complex genesis of faults, a genesis which in many circumstances is not captured anywhere in the software record.

What about reading previously written code caused the programmer to identify and fix this other error? How did the two areas of the code relate? How did tiredness factor into the later error, what were the circumstances, and did this have other effects on the software?

To see this discussion in context, please see also the section Implications for Future Research in the literature review, above. Please also note, the image in this section is of poor quality. The image in the original paper, while also blurry, is more readable, and the reader is directed to that version as required.
Appendix F. Guindon’s Kinds of Expert Knowledge

In the paper Knowledge exploited by experts during software system design, Raymonde Guindon (1990) analysed the specialised knowledge used by software designers when performing early design tasks. Her analysis included information about the kinds of new knowledge generated, the ways in which designers leverage existing knowledge, and a set of heuristics used in seeking and selecting design solutions. Her work built upon the findings of other design studies and software engineering research. In subsequent years, her discoveries have been integrated into design cognition studies (Cross, 2001). Guindon’s own contributions remain relevant in their own right, however, as do her particular articulations of the kinds and qualities of knowledge used in software design.

Guindon’s findings from this paper were extracted and consolidated into a catalog as an aid to analysis of early design activities in the SPSD sessions. More information
about this preliminary study can be found in the section Preliminary Study: Working Through Design Difficulties above.

They are enumerated according to the section of that paper in which they appear.

**Sect. 3.1** Retrieval or simulation of scenarios in the problem domain (the real world). Interwoven with solution development throughout the design session, the spoken scenarios are often accompanied by external representations in the form of diagrams with annotations.

Scenarios are used for five purposes:

1. Understand given requirements - before problem solving, as a way of confirming understanding of requirements.
2. Understand inferred requirements - upon inferring requirements, as a way of confirming the relevance of the discovery.
3. Solution development - to generate new ideas, to jumpstart progress. When used in this way, the scenarios are used to frame and structure the problem.
4. Discovery (unplanned) of new requirements - used to simulate and evaluate the solution.
5. Discovery (unplanned) of partial solutions - the scenario triggers the recognition of a partial solution.

**Sect. 3.2** Requirements elaboration, used to reduce ambiguity inherent in the design prompt and to decrease the range of possible solutions by acting as "simplifying assumptions" (p. 290). Running throughout the design session, requirements elaboration structures and frames the problem, and suggests evaluation criteria for solution selection. External representations of the requirements in the form of lists of notes are used to "keep track" of requirements. These notes help support systematic and balanced development, but do not indicate that the overall session might be so characterised.

1. Inferred constraint - unstated in the given requirements, but are inferred as logically necessary based on what is stated, and the designer’s own knowledge of the problem domain. They reduce incompleteness and ambiguity in the stated requirements, with direct consequences for the solution. In design sessions, they often result in changes in immediate design goals. That is, the designers shift the focus of their thinking to handle the newly inferred requirement.
2. Added requirement - a desirable but not necessary requirement for the production of a logically sound design. They reflect preferred evaluation criteria, or rules by which designers signify stopping points.
Sect. 3.3 Design Solutions, including the designer’s understanding of the solution, and the way this understanding is externally represented using particular design methods and notations. Guindon found that the way a solution was decomposed into subproblems may vary between designers, as may the selection of notational systems for representation. In general, she observed Uses of external representations:

- to express the design solution
- support mental simulations of the solution in the form of ”test cases” based on knowledge of the problem domain. Mental simulations are used to uncover various kinds of ”bugs” in the solution:
  - inconsistencies within given or inferred requirements
  - inconsistencies between parts of the solution
  - incompleteness of partial solutions in respect to the whole
  N.B. Guindon states a fourth, but it seems to be a duplication of an earlier point
- reveal missing information
- ensure completeness of the solution

Notational systems serve two purposes:

- express the design solution
- tools for developing the solution

Sect. 3.4 Design strategies, methods and notations, that is, the sequence of activities to be performed, as structured by a recognized design method. Guindon found that extracting these methodologies from the protocols was not difficult, as they manifest particular behaviours, and are often referred to by developers. Examples of design strategies given are: top-down, data structure-oriented and object-oriented structure. Designers can use more than one strategy in a single session, and may also use multiple notational systems.

Sect 3.5 Problem solving and software design schema, or higher order knowledge structures such as divide-and-conquer and generate-and-test. Guindon found that in her data, specialized schema used by designers varied in complexity and granularity. She suggested that the schema is a ”complex rule composed of a pattern which specifies the similarities in requirements between different instances of a class of systems (e.g. resource allocation systems).” (p. 296). Schema are selected based on similarities between the current problem and known patterns.

Sect. 3.6 Design heuristics are used by designers in problem structuring and solution generation. Guindon’s research observed the following heuristics were used most frequently:

(1) consider a simpler problem first, then later expand the solution
(2) simulate scenarios in the problem domain to acquire more information about the problem structure
(3) identify system functions that can be performed nearly independently and divide the system into corresponding subsystems
(4) concentrate on avoiding serious mistakes or catastrophes
(5) concentrate on satisfying the most important constraints or requirements first
(6) keep the design solution as simple as possible
(7) make reasonable simplifying assumptions about the requirements
(8) keep the solution parts as consistent as possible
(9) delay commitment to decision when there is insufficient information; re-examine tentative decisions as new information is acquired.

Sect. 3.7 Preferred evaluation criteria are adopted in order to manage the ill-defined nature of design problems. Guindon found that designers adopted a ”small set of personalized criteria” (p. 298) to guide solution generation and selection. For example, one of her developers adopted high reliability as a criterion. Unstated in the requirements, this criterion was used in schema selection, and thus to reduce the set of possible designs to consider. Other observed criteria included simplicity of solution and simplicity of design process.
References


Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. (2010). Exploring the influence of identifier names on code quality: an empirical study. This is a Phd candidate at the Open University; has created a dictionary of words potentially used in identifier names; look at the FindBugs software.


Flanagan, J. (1954). The critical incident technique. *Psychological bulletin*, 51(4):327. This is the seminal reference for the technique. Should read it more carefully, but as Chell notes, assumes a positivist stance, and so probably later evolutions will be more appropriate sources of direct information about methodology.


