INFORMATION, ACTIVITY AND SOCIAL ORDER IN DISTRIBUTED WORK: THE CASE OF DISTRIBUTED SOFTWARE PROBLEM MANAGEMENT

Robert John Sandusky, Ph.D.
Graduate School of Library and Information Science
University of Illinois at Urbana-Champaign, 2005
Les Gasser, Advisor

Results from a study of a global Free / Open Source Software (F/OSS) development organization show how distributed information practices contribute to the management of software problems. The study addressed these questions: (1) How do F/OSS development communities manage software problems? (2) What varieties of information and activity occur as part of software problem management? And (3) How are information, activity, and social order related in software problem management?

A systematic random sample of 385 bug reports was drawn from a collection more than 182,000 bug reports held in this community’s bug report repository, with additional bug reports selected from the repository using theoretical sampling. The bug reports were treated as texts; content analytic and grounded theory procedures were used to develop categories of phenomena and new and revised theoretical constructs. The analysis of the data was informed by previous research into human information behavior, negotiation, collaboration, social and information contexts, software engineering, and sensemaking to provide insights into how this large, distributed organization utilizes information to produce successful, high-quality software.
Contributions from this project include the first detailed, empirically grounded description of the information practices used by a distributed F/OSS development organization to manage software problems, including the first description of a fundamental information object, the bug report network; clearly drawn distinctions between the information management work necessary to support the community’s software problem management goals and the work that directly resolves software problems; identification of the processes and social arrangements that enable this community’s distributed collective software problem management practices; explication of software problem management as a distributed, communal sensemaking enterprise; description of the layers of context and nested processes necessary to support distributed, information-intensive work; and the description of a new class of human information behavior, distributed collective information practices. Throughout, the ways in which information, activity, social order, context, and process are mutually constitutive are identified and explained. Bug report networks are used as a primary example of how activity leads to new forms of information, re-orders existing information, affects social order, and contributes to the community’s efforts to bring problematic situations to closure.
INFORMATION, ACTIVITY AND SOCIAL ORDER IN DISTRIBUTED WORK: THE CASE OF DISTRIBUTED SOFTWARE PROBLEM MANAGEMENT

BY

ROBERT JOHN SANDUSKY

B.A., Northern Illinois University, 1979
M.S., Northern Illinois University, 1983

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Library and Information Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois
ABSTRACT

Results from a study of a global Free / Open Source Software (F/OSS) development organization show how distributed information practices contribute to the management of software problems. The study addressed these questions: (1) How do F/OSS development communities manage software problems? (2) What varieties of information and activity occur as part of software problem management? And (3) How are information, activity, and social order related in software problem management?

A systematic random sample of 385 bug reports was drawn from a collection more than 182,000 bug reports held in this community’s bug report repository, with additional bug reports selected from the repository using theoretical sampling. The bug reports were treated as texts; content analytic and grounded theory procedures were used to develop categories of phenomena and new and revised theoretical constructs. The analysis of the data was informed by previous research into human information behavior, negotiation, collaboration, social and information contexts, software engineering, and sensemaking to provide insights into how this large, distributed organization utilizes information to produce successful, high-quality software.

Contributions from this project include the first detailed, empirically grounded description of the information practices used by a distributed F/OSS development organization to manage software problems, including the first description of a fundamental information object, the bug report network; clearly drawn distinctions between the information management work necessary to support the community’s software problem management goals and the work that directly resolves software problems; identification of the processes and social arrangements that enable this community’s distributed collective software problem management practices; explication of software problem management as a distributed, communal sensemaking enterprise; description of the layers of context and nested processes necessary to support distributed, information-intensive work; and the description of a new class of human information behavior, distributed collective information practices. Throughout, the ways in which
information, activity, social order, context, and process are mutually constitutive are identified and explained. Bug report networks are used as a primary example of how activity leads to new forms of information, re-orders existing information, affects social order, and contributes to the community’s efforts to bring problematic situations to closure.
To my family
ACKNOWLEDGEMENTS

This project would not have been possible without the support and encouragement of many people: my family, my friends, and my colleagues in the Graduate School of Library and Information Science. Lora, Charlie, and Kiera have supported me in every way throughout the process, being willing and eager to try new cities, schools, and circumstances. We have held together despite my distraction and absences from their life. This dissertation is in many ways theirs as well as mine.

My mother and father continuously encouraged me and supported me throughout my academic and professional careers, for which I am grateful.

Les Gasser, Geof Bowker, Mike Twidale, and Walt Scacchi, my committee members, were instrumental in helping me see this project through to completion. Geof Bowker, my advisor during the early years, was unfailingly enthusiastic which gave me boosts of confidence at many critical points. Les Gasser, my advisor throughout my dissertation phase, provided me with the opportunity to pursue this project which was a perfect fit with my background and interests. Both Geof and Les provided encouragement, ideas, challenges, and support throughout. Other people at the Graduate School of Library and Information Science and in other parts of the University who made a difference for me include Leigh Estabrook, Linda Smith, Leigh Star, Patricia Jones (Mechanical and Industrial Engineering), Chip Bruce, Karen Ruhleder, Bruce Schatz, and my fellow students Gabriel Ripoche, Kevin Powell, Laura Neumann, Cece Merkel, and Emily Ignacio. Finally, I thank Ann Peterson Bishop, who, in addition to being my mentor during the Digital Library Initiative days, was my friend and my coach throughout the hardest parts of the process.

Some of the data collection, analysis, and presentation activities upon which this material is based were supported by the National Science Foundation ITR (Digital Society and Technologies) Program under Grant No. 0205346. Any opinions, findings,
and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.
# TABLE OF CONTENTS

**LIST OF TABLES**..............................................................................................................xii

**LIST OF FIGURES**................................................................................................................xiii

**CHAPTER ONE: INTRODUCTION**........................................................................................1
  Software Problems and Software Problem Management.....................................................4
    Impact of Software Problems..............................................................................................4
    The Software Problem Management Process......................................................................9
  Problem Statement..................................................................................................................15
  Research Questions...............................................................................................................18
  Research Approach...............................................................................................................21
  Benefits of the Research......................................................................................................23

**CHAPTER TWO: LITERATURE REVIEW**...............................................................................26
  Introduction..............................................................................................................................26
  Software Quality and Free / Open Source Software Development .....................................27
    The Importance of Software Quality................................................................................27
    Software Quality Research.................................................................................................27
    Organizational Structure, Software Process, and Quality..................................................30
    The New Silver Bullet: The Free / Open Source Software Process..................................34
    Free / Open Source Software Research............................................................................37
  Software Problem Management Research..........................................................................39
  Approaches to the Study of Information Practices in Distributed Work..............................43
    Information Needs, Uses, Behavior, and Practices............................................................43
      Collective and Constructive Information Practices..........................................................45
      Studies of a Wide Range of People and Settings..............................................................47
      Contextualized Inquiry into Human Information Behavior.............................................50
      User Centered Inquiry........................................................................................................52
    Negotiated Order................................................................................................................53
    Trajectories...........................................................................................................................54
    Situated Action Theory.........................................................................................................56
    Activity Theory.....................................................................................................................58
    Distributed Cognition..........................................................................................................59
    Organization Type and Information Practices....................................................................61
    Sensemaking.........................................................................................................................62
    Coordination/Cooperation Theories.....................................................................................68
  An Approach to the Study of Information Practices in Distributed Work............................70

**CHAPTER THREE: METHODS**............................................................................................78
  Introduction..............................................................................................................................78
Distributed Work is Shaped and Influenced by Interrelated Layers of Context

Future Directions

Future Research Program

Bug Report Networks

Basic Social Process: Negotiation

Information Object State Sequences

Deeper Investigation of Information Practices

Distributed Collective Ensembles

Context and its Relationship to Distributed Work

Process and Distributed Work

Application of Additional Data Sources and Additional Research Methods

Application of Findings to Tool Development

Toward a Theory and Model of Distributed SWPM

LIST OF REFERENCES

APPENDIX A: CODE LIST

CURRICULUM VITAE
LIST OF TABLES

Table 1-1 Normative Software Problem Management Process ................................................................. 12
Table 3-1 Bug Report Population: Distribution by Most Frequently Occurring Product Values .................. 84
Table 3-2 Bug Report Sample – Distribution by Product by Month ........................................................... 90
Table 3-3 Comparison: Sample to Population, by Product .................................................................... 92
Table 4-1 Examples of Statement Types from Bug Reports ...................................................................... 115
Table 4-2 Bug Report Status and Resolution Values ................................................................................ 120
Table 4-3 Bug Report Completion States .................................................................................................... 123
Table 4-4 Bug Report State Sequences: Patterns and Frequencies ............................................................ 123
Table 4-5 Question and Answer Patterns ................................................................................................. 126
Table 5-1 Frequency of BRN relationships .................................................................................................. 136
Table 7-1 Normative and F/OSS Software Problem Management Processes ............................................ 172
Table 7-2 Phenomenon / Encounter / Bug Report Combinations ............................................................ 176
Table 7-3 Disposition of Bug Reports During Triage ................................................................................ 182
Table 7-4 Frequency of Negotiation in Sample .......................................................................................... 195
Table 7-5 Issue Frequency in Sample ......................................................................................................... 196
Table 8-1 Formal and Informal Roles in Distributed Cooperative Ensembles (DCEs) ................................. 218
Table 8-2 DCE for BR-E ............................................................................................................................ 219
Table 8-3 DCE for BR-A (Meta bug report) .............................................................................................. 219
Table 8-4 DCE for BR-B ............................................................................................................................ 220
Table 8-5 DCE for BR-C ............................................................................................................................ 220
Table 8-6 DCE for BR-D ............................................................................................................................ 220
Table 8-7 DCE for BR-F ............................................................................................................................ 221
Table 8-8 DCE for the Bug Report Network Comprised of BR-A .. BR-F ..................................................... 222
LIST OF FIGURES

Figure 2-1 Model of Human Information Behavior / Information Practices.............................. 75
Figure 4-1 Structure of Potential Analytic Units ............................................................................. 101
Figure 4-2 Anatomy of a Bug Report ................................................................................................. 108
Figure 4-3 Anatomy of a Bugzilla Bug Report: Structured Fields and Input Areas ...................... 118
Figure 4-4 Anatomy of a Bugzilla Bug Report: Comment Area ..................................................... 119
Figure 4-5 Anatomy of a Bugzilla Bug Report: Activity Log ............................................................ 119
Figure 5-1 Bug Report Repository Patterns ...................................................................................... 131
Figure 5-2 Bug Report Network ........................................................................................................ 134
Figure 7-1 Processes and Sub-Processes in Distributed Software Problem Management............. 171
Figure 8-1 Text of BR-E ...................................................................................................................... 215
CHAPTER ONE: INTRODUCTION

Issues of creating, organizing, seeking, finding, using and sharing information are central concerns of workers, managers, students, citizens, information systems designers, educators, policy makers and researchers in many domains, including medicine, education, commerce, government, communication, the arts, science and engineering. These information practices are employed in support of decision making, informing action, reducing complexity and helping make sense of uncertain or problematic situations. Changes in the technical capabilities of information systems, forms of information, and supporting infrastructure continuously affect the information practices of people, organizations and society. One of the most significant technological changes in recent years, beginning in earnest during the 1960s, is the deployment of ubiquitous standardized digital communications networks. These networks have enabled the evolution of existing and emergence of new kinds of information practices during the intervening decades. Information, infrastructure, tools, and information practices have become increasingly distributed and less dependent on tight synchronization of activities and the co-location of information and its users.

This project, employing a holistic, social informatics approach, uses software problem management (SWPM) work as an exemplar of distributed work to address questions about the interactions between information, activity, and social order, and to help us understand how distributed organizations utilize information to actually “get things done.” This project applies ideas from research into human information behavior, negotiation, collaboration, sensemaking, and software engineering in order to understand the information practices of a complex distributed software development organization. While research into software defects has been done, primarily from software engineering or psychological perspectives, this project is unique because it looks at software problem management as a complex, organizationally situated socio-technical process. This project takes a particularly close look at the information behavior of individuals and small groups
in order to identify and understand the information practices used by this distributed organization.

The continuously changing technical and social landscape invites regular investigation into information practices in all settings. Information practices evolve as changes are made to any part of the “inseparable ensemble” (Kling, 2003) of technology, information form, social order, process, context or activity. Individual and group behavior, organization practices and culture, technology and information must all be taken into account in order for research to contribute to our understanding of fundamental human information behavior in complex, distributed settings.

A case study approach is used in this project to examine the software problem management work undertaken in one large, distributed free / open source software (F/OSS) development community as those practices are revealed by the records contained in that community’s bug report repository. I analyzed a systematic random sample of 385 bug reports, drawn from a collection of nearly 200,000 bug reports held in the bug report repository maintained by this thriving F/OSS development community. Using content analytic and grounded theory techniques, I iteratively identified information, activity, processes and forms of social order; generated concepts and categories of concepts; and identified relationships between them as I analyzed the bug report sample. The approach taken in this project is consistent with the tenets of social informatics: complex information systems and practices related to information management and information use should be examined in a way that takes organizations, people, information, and technology equally into account, treating those elements as an inseparable ensemble (Kling, 2003).

This project addresses these questions: (1) how do distributed software development communities manage software problems? (2) what kinds of information practices occur as part of the distributed software problem management process? and (3) how are information, activity, and social order related in distributed software problem management? The goals of this project are to develop an empirically based, comprehensive map of distributed SWPM work, to analyze this kind of distributed work
in terms of existing theory and methods, to extend existing theory, to lay the groundwork for the specification of a theory of software problem management, to generate testable hypotheses that support that theory, and contribute to the development of tools and techniques that improve the production of software. It is anticipated that future research will refine and generalize these findings, and thus contribute to our understanding of information practices in a wide range of distributed settings in a variety of domains.

This dissertation makes the case that software problem management as a whole can be effectively described as a sensemaking endeavor (Weick, 1995). In software problem management, the bug report and the bug report repository provide a common frame of reference that the community uses to turn puzzling phenomena, software problems, into bug reports, information objects with a widely-understood format and function. A bug report is a problem representation that community members can work on as a community; in effect, the bug report provides the community with a problem that can be solved by applying the software problem management process, the community’s sensemaking apparatus. The process of managing software problems and bug reports – sensemaking – depends upon the information behaviors of individuals working in small groups in combination with the entire community’s information practices. Bug reports (by design and through the evolution of practice) simultaneously structure and hold information. They provide a selective record of activity while also directing future activity. The community can also compose bug reports into bug report networks, more complex entities that, through their very structure, convey new information. Bug reports, in combination with the bug report repository, embody and thereby privilege certain processes and map to attributes of the embedding social order.

This introductory chapter provides background information about the domain, states the research problem, identifies the research questions addressed by this project, discusses the approach taken to address the research questions, and summarizes the contributions of this project. The literature review places this project in the context of previous and current research in traditional and distributed information systems development practice and discusses a number of possible approaches to understanding
information use and activity in distributed work settings. The third chapter describes the methods used to gather and analyze the data in this project. The chapters that follow describe and analyze the findings from the analysis of the data. Chapter Four presents the variety of potential analytic units present in this data. The fifth chapter discusses a vital distinction between phenomena (software problems) and information objects (bug reports) that represent those phenomena and an important sensemaking strategy the community employs to simultaneously restructure information and the social order. Chapter Six argues that layers of context play a vital role in shaping the information forms, activities, processes, and social order that emerge in this distributed work community. The seventh chapter focuses on processes: the role of negotiation in the coordination of activity, the active structuring and creation of information and the evolution of negotiated order in the community. The critical importance of bug report triage is also examined. Chapter Eight looks at social order in this distributed community and proposes the use of roles as a method for understanding the dynamic nature of social order in this community. The final chapter summarizes the findings and contributions of this dissertation and identifies directions for future research.

**Software Problems and Software Problem Management**

**Impact of Software Problems**

Software problem management work is intrinsically interesting: software is increasingly ubiquitous and complex, a key part of constellations of tightly and loosely coupled socio-technical systems. Defects in this software are recognized as having enormous financial, social and human impacts.

Software problems (also often referred to as bugs or software defects) are errors or mistakes of commission or omission inadvertently introduced to software during the software development process (Gasser, 2003). Kajko-Mattsson (1999), quoting Florac (1992) (p.24), defines a software problem as “a human encounter with software causing a
difficulty, doubt, or uncertainty” (Kajko-Mattsson, p.169. Humphreys prefers the term “software defect:

The term defect refers to something that is wrong with a program. It could be a misspelling, a punctuation mistake, or an incorrect program statement. Defects can be in programs, in designs, or even in the requirements, specifications, or other documentation. Defects can be redundant or extra statements, incorrect statements, or omitted program sections. A defect, in fact, is anything that detracts from the program's ability to completely and effectively meet the user's needs. A defect is thus an objective thing. It is something you can identify, describe, and count. (Humphreys, 2004)

Software problems vary in nature and can be introduced at any stage of the development process. Software problems are continuously identified – and bug reports created – while the members of the development community design and create new software. The work to manage the flow of new bug reports and fix the problems represented by the bug reports is continuous, too, because it is an intrinsic part of software development. Bug reports reveal the nuances of software problem management, problem trajectories, problem contexts and their relationships to process and social order within any software development community.

Software problems are an important research topic because of their impacts: software problems are costly to find and repair; high defect rates can affect the viability of organizations that create and sell software; use of software with a high defect rate can affect the viability of organizations that purchase and use the software; human safety can be compromised by low quality software used in critical applications like air traffic control systems and medical instruments.

A recent study by the National Institute of Standards and Technology on software testing infrastructure estimates that $22-60 billion dollars are lost in the U.S. economy each year due to inadequate infrastructure for detecting and correcting software problems, and the subsequent release of bug-ridden software to users and consumers
(National Institute of Standards and Technology, 2002) (p. ES-3). This cost estimate includes both the costs to developers to repair problems and the costs to user organizations to mitigate their effects. Another study estimates that removing the typical 5-15 bugs per 1,000 lines of code takes as long as 150 hours and costs as much as $30,000 (cited in Gross, et. al, 1999). 97% of the organizations responding to an industry survey indicated that their company had “used software in the past year that has produced bugs, errors, or other vulnerabilities” (Hayes, 2003) (p.50).

In addition to the costs of finding and fixing these relatively prosaic bugs, software problems can also have significant social and human impacts. In the mid-1980s, faulty software used to control a radiation therapy machine caused at least six accidents resulting in deaths and other serious injuries (Leveson & Turner, 1993) and again, in a separate situation, in 2001 (Gage & McCormick, 2004). False warnings generated by faulty safe altitude warning system software led air traffic controllers on Guam to drastically reduce the range of the warning system, from 54 miles to 1 mile. Controllers were then unable to warn the pilot of KAL flight 801 that the plane was about to crash into a mountain, killing 225 people (Gage & McCormick, 2004; National Transportation Safety Board, 2000).

Software problems have caused several notable disruptions to telecommunications facilities. One of these outages occurred on January 15, 1990 on the AT&T long distance telephone network. This bug manifested itself when, under certain conditions, network switches cycled through a normal recovery sequence and signaled their status to neighboring switches. The bug made the neighbor switches vulnerable to going through the same recovery cycle, resulting in switch failures that cascaded, or traveled, across the long distance network. An estimated 70 million telephone calls were prevented from completing during the nine hours technicians worked to resolve the problem (Risks Digest, 1990).

---

1 This dollar estimate excludes costs “…associated with mission critical software where failure can lead to extremely high costs such as loss of life or catastrophic failure.” (NIST, 2002) (p. ES-3).
Other problems have occurred in telephone systems (in 1999 an 8-hour outage in Stockholm, Sweden affected millions of telephone lines and prevented communication by phone to the police department; two-thirds of Singapore’s telephone lines were knocked out of service in 1994); Internet routing systems (in a 1997 incident, MAI Network Services supplied incorrect routing tables to Internet backbone service providers); missile guidance systems (missiles missed targets during the 1991 war with Iraq). Many more examples of these kinds of problems can be found at Neumann’s Web site dedicated to new risk-related material at http://www.csl.sri.com/users/neumann/risks-new.html.

Perhaps the most commonly understood kind of software problem is one where a functional computing activity, like adding up a column of numbers or saving a computer file, does not work correctly. Other kinds of software problems are less obvious and perhaps more common. Some examples (organized by ISO-9126’s software quality attributes (International Organization for Standardization, 1991)) are (National Institute of Standards and Technology, 2002) (p.1-5):

Functionality:

- During the software design process, user requirements are misinterpreted or overlooked.
- Computer standards for transaction format, document format or network communication are implemented incorrectly.
- Interactions between different parts of a large software system, designed and coded by many different people, produce incorrect results.

Reliability:

- A software system requires frequent restarting while the user is performing tasks, causing interruptions and wasting time.
- Bugs in a system’s security software lead to financial loss or theft of an individual’s identity.

Usability:

- The user interface is inconsistent, difficult to understand or does not support common user tasks in a straightforward way.
• The software is difficult to use because it implements a process model that doesn’t match the users’ processes.

Efficiency:

• Users may be able to correctly perform a function only by taking care to work around, or mitigate, system deficiencies.
• Elapsed time to return search results exceeds requirements.

Maintainability:

• Modifying the software causes unanticipated side effects (bugs).
• Identification, isolation, and repair of bugs is difficult because of poor software design.

Portability:

• Versions of a program running on different computing platforms exhibit different behavior.

Some suggest that bugs should not occur and are a symptom of a poor process, a poor design or bad programming. Software problems are, however, normal and to be expected, particularly in complex systems (Perrow, 1984; National Institute of Standards and Technology, 2002). Software problems may be identified at any time during the software design, development and use lifecycle. Requirements and design problems may occur and may (or may not be) identified before programming begins. Functionality problems may be detected early in the development process or during formal testing (Carstensen, Sørensen & Tuikka, 1995). Users may detect problems after software is released to the market. Some problems may manifest unexpectedly under a set of unusual circumstances (like the AT&T long distance system or the radiation therapy machine problems described above).

If problems do occur, some may expect that the problems, once identified, should always get fixed. But software problems, even when identified and well characterized, don’t always get fixed. Examination of bug reports shows that significant
percentages of bug reports remain open and active for months or even years. Some bugs may exist that aren’t identified for a long period of time, and other bugs may never be identified. A number of factors can make software problems difficult to fix. The system may be designed or coded in a way that makes understanding the system difficult. Documentation on the system and its design may be non-existent or out of date. The people who developed the software may not be available for consultation by those expected to fix the bugs. Changes to fix a bug may introduce additional bugs. Bug reports – the descriptions of problematic behavior that drive the process – may be misleading or incorrect: if the problem behavior cannot be recreated, the bug is unlikely to be fixed.

The Software Problem Management Process

Software problem management is part of the larger challenge of software quality assurance that includes the development and application of development methodologies (from early structured methods to contemporary agile methods), development of software quality metrics (e.g., faults per thousand lines of code) and a variety of approaches to software testing. Software problem management (SWPM) is the process by which software problems are identified, analyzed and, hopefully (but not always) repaired. Software problems are expected to occur, and an organization’s SWPM process is designed to provide control over the way software problems are identified, reported and corrected. A defined and routinized SWPM process helps make problem correction repeatable through the codification of elements of the process. The artifacts and systems used to support the process provide a degree of structure and support individual and organizational accountability for correction of problems. Most bug report systems define distinct roles, like reporter, assignee, quality assurance, etc., in order to enforce the proper and open handling of reported bugs (Gerson & Star, 1986). Proper and open handling is desired because it ensures that changes to the software are reviewed by appropriate people and organizations; that changes are made by those most qualified to make them; problem status is widely available; problems are addressed using a standard, routine
process. Use of a standard process is expected to reduce the costs associated with solving problems and helps ensure that the quality of the software doesn’t slip (Crowston, 1997).

Bug reports are a central artifact in any non-trivial software development community. Bug reports have a life cycle: according to the normative SWPM process described below, they are created, remain open (or active) for a period of time and are eventually closed. This project will show, however, that there are many more possible bug report and software problem trajectories than are described by the normative model. Bug reports represent many different things and are a nexus of information and activity while active. The bug report makes the problem visible to other people and organizations and helps to coordinate a chain of activity that, according to normative models, should result in the fixing of the bug. The bug report carries a series of annotations that describe past events and point to future work. A bug report contains descriptive information about the bug. The bug report supports re-creation of the problem situation so that the person(s) or organization(s) responsible for fixing the bug have sufficient information to develop and test a resolution. Bug reports contain information about what work has been done, what work is underway, and what work is planned or committed to in the future. A bug report records who has been involved in the reporting, triage, analysis, coding, and verification of the bug and, hopefully, its fix. Bug reports timestamp changes made to the bug report and identify who made each change. Bug reports often also contain opinion, hypotheses, clarifications, and conjecture, entered as text and made available to anyone in the community. By closely examining bug reports, we can get a sense of the activities and processes employed; the kinds of information gathered, used and created; the parts of that process that work and the parts that go wrong, and how bugs come (or fail to come) to closure.

The bug report comments and the bug activity data also form a representation of a bug report’s trajectory – a path, progression, or line of development resembling a physical trajectory – from the time the bug report is created until it is set to a terminal state (e.g., “status = verified”). The bug report also reveals information about the community’s software problem management processes and social order; how the work of managing
software problems is aligned and coordinated; who participates in that process; how the organization of the community affects and is affected by the information contained in the bug report.

Crucial to understanding SWPM is the distinction between a software problem and a bug report. A software problem is a phenomenological situation, generally lacking tangible form, in which software system behavior fails to conform to some group’s expectations and values in a manner judged to have negative sentimental, moral or practical import (Gasser, 2003; Kajko-Mattsson, 1999). A bug report is created to represent a software problem, and functions as a formal and tangible representation of the problem. In different situations, members of the community may do work on either a software problem or on a bug report. If a programmer is making and testing code changes, he is working on a software problem. If a manager is evaluating whether a new bug report is a duplicate of another existing bug report, she is reading, evaluating, managing, and linking bug reports, or records, in the bug report repository. The distinction between bug reports and software problems is critical to our analysis because typical collective repository tools support bug reports as first class objects, but the aims of analysis and action are software problems, which are not first-class objects in repository tools. For example, many bug reports may contribute to the understanding or resolution of a single software problem (Gasser & Ripoche, 2003; Sandusky, Gasser & Ripoche, 2004a). The distributed, collective activity of SWPM is mediated in a development community by information and communication technologies. Collective interpretations and actions (such as judgments on causality, management actions and code creation) are actively constituted, constructed and coordinated via social processes such as negotiation, and the traces of negotiation in bug reports serve as surrogates for plans, interpretations, formal specifications, decisions, rationales, etc.

Empirical observations made as part of this research project reveal that bug reports often represent phenomena or entities other than bugs. For example, bug reports may represent requests for new features, bug report networks (Sandusky, Gasser & Ripoche, 2004a), arbitrary collections of bug reports (“meta-bugs”), invalid problems
(that is, issues that the community decides are not going to be worked on), duplicate reports of the same phenomenon, and even social gatherings (e.g., a party).

The description of the software problem management process (Table 1-1) includes references to activity, information and representations of problems (bug reports). The bug report may be created in any of a number of ways depending upon the conventions adopted by a particular organization: it may be written on a paper form but is usually entered into a database. Whether it is a binder containing sheets of paper or a relational database management system, I refer to such a database as a bug report repository. As bug reports are the central artifacts, bug report repositories are the central information system supporting software problem management. The community uses the bug report repository to manage both bug reports and software problems. The bug report repository is used to keep the work and activity of many people working on thousands bug reports coordinated and prioritized and contains much of the information needed to fix software problems.

Components of the process vary depending upon the organizational context and complexity, but the following steps comprise a typical process. Carstensen, Sørensen & Tuikka’s (1995), Schmidt & Simone (1996), Crowston (1997), and (Kajko-Mattsson, 1999) in particular provide information that informs this normative process model.

<table>
<thead>
<tr>
<th>Table 1-1 Normative Software Problem Management Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>Problem Identification</strong></td>
</tr>
</tbody>
</table>

Someone using the software notices an anomaly or mistake; identification can occur during normal use or during software testing; actors identifying problems may be testers (Carstensen, Sorensen & Tuikka, 1995; Crowston, 1997) or customers / users (Crowston, 1997).
2. Problem Identification

Someone using the software notices an anomaly or mistake; identification can occur during normal use or during software testing; actors identifying problems may be testers (Carstensen, Sørensen & Tuikka, 1995; Crowston, 1997) or customers/users (Crowston, 1997).

3. Problem Reporting

Someone creates a bug report—a representation of the software problem—to enable the software problem and follow-up activities to be tracked through a bureaucratic process; actors creating the bug report may be testers (Carstensen, Sørensen & Tuikka, 1995) or an intermediary such as a help desk on behalf of the customer or user (Crowston, 1995).

4. Bug Report Triage and Assignment

Someone takes action to begin the process of identifying and fixing the software problem. An attempt to recreate the problem is made. The priority of the bug report is assessed (triage); the proper person or organization to work on the bug is identified (assignment). In Carstensen, Sørensen & Tuikka’s study (1995), priority and assignment are handled by a group of software designers whose role is the management of software problems. In Crowston’s study (1997), set in a larger organization, problem recreation and prioritization is done by “marketing engineers” and assignment by “programming managers.”

5. Expert Analysis, Fix Development, Testing, and Deployment

The assignee investigates the software problem, determines its cause, evaluates repair options, consults other experts, coordinates the work of multiple experts, etc. The assignee modifies the software to resolve the software problem and tests the modification. The modification, or fix, is deployed if testing is successful. Software designers (Carstensen, Sørensen & Tuikka, 1995) or software engineers (Crowston, 1997) usually perform this task.

6. Fix Verification

Someone (who usually plays a quality assurance role) other than the assignee verifies that the software problem has been corrected. The platform master (the person responsible for the software build containing the fix) (Carstensen, Sørensen & Tuikka, 1995) or the integration team (Crowston, 1997) may take on this role.

---

2 Bug report repositories are also referred to as “bug tracking,” “defect tracking,” “issue tracking” or “problem management” systems.
Table 1-1, cont.

<table>
<thead>
<tr>
<th>Normative Software Problem Management Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>7. <strong>Problem Closure</strong></td>
</tr>
<tr>
<td>Someone marks the bug report “closed;” the process is complete. In Carstensen, Sørensen &amp; Tuikka (1995), this is the responsibility of the central file manager; in Crowston (1997), this is the software engineer’s responsibility (Crowston, 1997).</td>
</tr>
</tbody>
</table>

The first step in the process, problem identification, occurs when someone using the software notices behavior that they consider undesirable or abnormal. Alternatively, the user may fail to see something he expects or something she would like to see: that is, the software is either missing functionality or has implemented functionality in an unanticipated way. In step two, reporting, the person who noticed and reported the software problem (the reporter) usually creates the bug report. During triage and assignment, the third step, someone with special privileges related to the management of bug reports (e.g., a quality assurance worker), examines the bug report and makes judgments about priority relative to other bug reports and who should be assigned to fix the problem. This person may also attempt to confirm that the problem can be recreated, or caused to manifest, at will by following the description provided by the reporter.

The fourth step, expert analysis and fix development, testing and deployment, are the stages when the “debugging” process most people are familiar with occurs: developing hypotheses, creating and testing fixes and finally deploying the fix into the software. The assignee is typically someone expert in, or at least familiar with, the software or software sub-component presumed responsible for causing the problem. The assignee typically has privileges to modify the status of any bug report assigned to him, but cannot modify bug reports assigned to others. He or she is also probably not able to perform bug report triage and assignment or bug verification and closure, which are the province of the quality assurance staff.

---

3 Someone else may actually cause the creation of the bug report. The reporter might use the telephone to report a bug: the person handling the call actually enters the reporter information on behalf of the reporter.
Fix verification, the fifth step, is part of the quality assurance process: someone other than the developer independently tests the fix and either agrees or disagrees that the problem has been resolved. The reporter is sometimes involved during the verification process: the reporter may be required to agree that the problem is fixed. The final step, closure, when present, represents a final bureaucratic checkpoint. Some organizations, for example, publicly review recently resolved problems to ensure critical process requirements are met: documentation is complete; code reviews were performed, etc.

The normative SWPM process, however, fails to account for some of the situations that have been identified in the F/OSS development community studied in this project. “Meta” bug reports, for example, are frequently created to support the collocation of bug reports having arbitrary characteristics in common (e.g., all of the critical priority bug reports that must move to resolved status before the next software release). “Meta” bug reports of this kind don’t represent software problems at all, so it is not possible to ever “resolve” what they represent. Yet the community must somehow manage these bug reports. Other shortcomings of the normative SWPM process described above will be noted in subsequent sections of this dissertation.

**Problem Statement**

The causes of and potential cures for the generally low level of software quality continue to be concerns of researchers and practitioners in computer science, applied computing, social informatics and business. Software development (the building of new systems) and software maintenance (the introduction and management of changes to software, often as a result of changes in the needs of system users) as socio-technical processes have been subjects of research. As early as 1970 it was recognized that “programming is not just a matter of hardware and software” – computer programming could be studied as a form of human activity (Weinberg, 1971) (p. vii). Other early work addressed the issues raised by alarmingly common software development project disasters: systems that, when delivered, did not meet the needs of users; systems that were
delivered far past deadline and far over budget; systems that were never completed at all. Debate and research occurred about issues such as (1) how software projects should be organized and managed; (2) how software design could most effectively be expressed in artifacts so that users, analysts, designers, coders, testers and technical writers could come to agreement on what needed to be built and agree on when a system meeting those needs had been completed. (See Brooks (1995) for an influential and readable discussion of such issues including the influence of project team organization on work and the effects of system and organizational complexity on project outcomes.) A software development methodology sub-industry arose during the 1970s and prospers today because the problems described above have still not been completely solved. Hirschheim, Klein and Lytinen (1995) provide a comprehensive overview and critique of several software development paradigms including the dominant “structured,” or functionalist, approaches as well as various socially informed approaches, like participatory design.

Software methodologists consistently argue that it is most economical and efficient time to identify and resolve software problems as early as possible in a project (McConnell, 2001). That is, it costs less to apply resources to fix problems earlier in a development cycle (e.g., during requirements analysis or design) than it is to fix problems later (e.g., during testing or after the system is delivered to the users). These cost patterns hold whether a waterfall or agile process is used in a given project even if the cost ratios decline (McConnell, 2001). Software methodologists sometimes talk about software problems, but usually with a focus on how an individual might approach and resolve distinct problems (suggested problem solving methods and heuristics) or on how the computer itself might help identify and support resolution of bugs (debugging tools or automatic defect detection). The larger issues of debugging and software problem management as complex socio-technical processes, however, are rarely addressed by research.

Software development and maintenance, like software problem management, is a socio-technical process. Fixing problems is technical work in that the computers and the associated tools (database management systems and compilers, for example) depend
upon application of specific technical knowledge and processes. The goal of the process is to produce a working software system that will be used in specific social settings, such as offices, by users with varying expectations, knowledge, and habits. The production of the software system also occurs in a specific social setting. Except in the most trivial cases, software problem management necessarily involves the coordination of work among people with a variety of backgrounds and skills: users (who are usually not computer scientists or software engineers) may report problems yet have little or no knowledge or experience in software development; quality assurance workers may have familiarity with the software development process but may not have significant programming skills; programmers may lack knowledge and experience that is typical of the user community. Other individuals – business analysts, visual or interface designers, database administrators, software developers, project managers, marketing and sales specialists – may become involved in the management of software problems. While working together on problems they are all likely to use the same bug report repository, and the repository must adequately represent and support their information needs and uses.

Software problem management work, particularly in large-scale settings, depends upon the effective organization and access to information. Software itself is information and thus the development and repair of software is information management. In some contexts, hundreds of bugs may be reported every day (in May, 2002, an average of 228 bug reports were created each day to the bug report repository used by the F/OSS development project studied here). The effective handling of such a large volume of information depends on technical systems (e.g., an effectively designed bug report repository) and social processes (e.g., an effective process for quickly triaging incoming bug reports). Classification systems are required to help organize thousands of bug reports into useful sub-collections; the bug reports themselves must be designed to carry appropriate information and be malleable enough to be useful in many settings, to many people with different backgrounds, interests and information needs. In this sense, bug reports and the repositories that contain them are boundary objects (Star, 1989).
Software problem management work, a central part of the process of creating and maintaining high-quality software, has received very little research attention. Software problem management work is of interest because of society’s enormous investment in and dependence on software. As discussed earlier, billions of dollars are spent on software every year in the U.S. alone. Software problem management significantly increases the cost associated with software (National Institute of Standards and Technology, 2002). The ubiquity of software in twenty-first century life in a wide range of mundane and safety-critical systems suggests that increasing the effectiveness of software problem management through knowledge gained through empirical research can have positive human, social, and economic impacts.

Research Questions

Bug report repositories maintained by F/OSS development communities are publicly available for study and provide an immense and rich source of data for researchers studying the nature of distributed work. This project provides fundamental knowledge about this process through the examination and analysis of the bug reports created by one F/OSS community as it continuously designs, implements, and refines its software products.

This project addresses the following research questions:

1. How do distributed F/OSS software development communities manage software problems?
2. What kinds of information practices occur as part of distributed software problem management?
3. How are information, activity, and social order related in distributed software problem management?
The first research question is addressed by using the empirical data, a large random sample of bug reports, to ground a detailed description of the process by which this F/OSS development community manages software problems.

The second research question is addressed by identifying a number of kinds of information, activity and processes present in the bug report data. In addition to noting the varieties that occur, this report will identify which are common and which are rare; when possible, which are “normal” and which are “deviations from the norm;” the degree of variance within the varieties and the ways in which that variance matters.

The final research question is addressed by identifying, where possible and where warranted, the relationships between activity, information and social order, and the possible ways information, activity, process, context and social order may determine the effectiveness of a community’s software problem management process.

Addressing these research questions results in the development of a grounded, empirically derived description of the information practices employed in distributed work, specifically in the domain of software problem management. The generation of such a description is a prerequisite to continuing organizationally situated empirical research into the information and work practices employed in distributed work in this and other domains. The development of generalized theories and models of distributed information practices, software problem management and software quality assurance can proceed only if it is based upon systematically derived empirical data. The third research question, concerned with the identification and development of relationships between phenomena identified by addressing the first two research questions, can only be addressed after an accurate “map” of concepts, processes, information, social order and activity is drawn.

The notion of developing a map of the information, activity and social order in this domain was present from the outset of this project. The kind of mapping I am attempting is akin, but not identical, to the kind of mapping associated with scientometrics, the application of bibliometric methods to the study of dissemination of knowledge in scientific fields (Callon, Law & Rip, 1986). I can to some extent map the
dynamics (or, “study the structure of research areas” Callon et. al. p.12) of distributed software problem management work by studying the texts (the bug reports) created by and left behind by the participants of the community:

A scientific text not only reveals the world-building strategy of its authors, but also the nature and force of the building blocks derived from the domain of science from which it draws and to which it contributes. The text thus provides access to the dynamics of science, to the shared worlds that constitute a means of mutual (and evolving) control. But how can such worlds of science be mapped? In principle, the answer is simple. The text may be treated as a structure linking important words. (Callon, et. al., p.12).

However, I am not taking a bibliometric approach, and not utilizing texts in the same way. Instead, starting with objects like a bug report repository, a collection of bug reports and various phenomena (e.g., software problems), I am working to discover what other objects and phenomena play a significant role in SWPM. During the mapping process, I have identified, analyzed and proposed relationships between previously undescribed forms of information (e.g., bug report networks), frequently employed social processes (e.g., negotiation), interrelated contexts (e.g., structural, technical, and negotiation contexts) and emerging social order (e.g., distributed dynamic cooperative ensembles).

The descriptive goals of this project are appropriate given the current level of knowledge about distributed information practices and work, software quality assurance practices generally and software problem management in particular. A thorough description of SWPM processes has not yet been completed and published. Previously published descriptions are relatively thin and focus on the normative aspects of the process in the particular settings from which they are derived. Until the range of information, activities, contexts and social order are carefully identified, categorized and mapped, the development and testing of hypotheses and the establishment of software problem management theory and models cannot be reliably attempted. The holistic social informatics outlook taken in the execution of this project helps to ensure that the
resulting map is not biased toward or away from the formal or the informal, the social or the technical, the organizational or the individual.

Future research in software problem management, software quality assurance and information practices in other kinds of distributed work can use this map as a guide, with subsequent projects taking a focus on providing more details to a particular part of the map (e.g., closer analysis of the sensemaking processes in software quality assurance); testing of hypotheses; development of scalable, automatic processes to foreground phenomena and relationships that are obscured by the large number of records in the repository and their obfuscation by the ambiguity of language.

At this point, a purely quantitative or hypothesis testing approach to the study of software problem management is not warranted: we do not yet know either the salient phenomena or the significant relationships between the phenomena that would justify such approaches.

Research Approach

The research questions listed above have been addressed by performing an empirical, grounded analysis of the bug reports created and managed by this F/OSS development community. Bug reports are the primary unit of analysis in this study. Bug reports and bug report repositories created and maintained by F/OSS development communities are publicly available, providing an abundance of data and containing the information recorded about the software problem’s behavior and the community’s actions and ideas about how the problem will be resolved.

This study also makes use of descriptions of the software repair process. Some of these descriptions, already noted above, are normative descriptions that, while situated in specific organizational contexts, do not reveal the full complexity of distributed software problem management work.

This study analyzes a systematic random sample of bug reports drawn from one F/OSS development community’s bug report repository. The documents were coded
and analyzed using grounded theory procedures. This process is described in greater
detail in the chapter on Methods, below.

This project examines what happens when people are engaged in distributed
work in the context of a new organizational form. Free / open source software
communities are highly interconnected, collaborative and distributed. This study takes a
social informatics approach to identify the varieties and contexts of software problem
trajectories and to identify the processes and social order that act on and are acted upon
in managing software problems. Social informatics is the study of the “…social aspects of
structuring and communicating knowledge in computer-based information systems.”
"…the interdisciplinary study of the design, uses and consequences of information
technologies that takes into account their interaction with institutional and cultural
contexts."

Kling emphasizes the importance of context (e.g., organization context) in
understanding information system usage.

One key idea of social informatics research is that the "social context" of information
technology development and use plays a significant role in influencing the ways that
people use information and technologies, and thus influences their consequences for
work, organizations, and other social relationships. Social context does not refer to
some abstracted "cloud" that hovers above people and information technology; it
refers to a specific matrix of social relationships. Here, social context is characterized
by particular incentive systems for using, organizing, and sharing information at work
(Kling, 1999).

A social informatics approach to the study of bug fixing takes the point of view
that technology, processes and social elements should be viewed as an “inseparable
ensemble” (Kling, 2003). Research into socio-technical systems may examine the ways in
which people act in “various roles and relationships with each other and with other
system elements; support resources (training/support/help); and information structures
(content and content providers, rules/norms/regulations, such as those that authorize
people to use systems and information in specific ways, access controls)” (Kling, 1999).
Social informatics, as an approach to research problems, does not privilege information, technology, the individual, the organization, activity, artifacts, processes or contexts. The approach sensitizes us to the complexity inherent in the study of information in authentic, and thereby complex, situations.

Managing software problems is a socio-technical issue: even with the best tools, software problems don’t fix themselves. The bug report repository is the system that links the community together, both supporting a particular negotiated social order and influencing the enactment of the social order as the software is designed, developed and repaired over time. The bug reports held in the repository are the artifacts, replicated thousands of times that provide a common frame of reference that can be applied in a useful way in both routine and exceptional situations.

**Benefits of the Research**

This project makes methodological, theoretical and direct practical contributions to our understanding of distributed work in general as well as information practices and software engineering. The data analysis yields detailed descriptions of the SWPM process in one large, successful F/OSS development project. This information can be used to orient investigations into other similar organizations or other kinds of organizations (e.g., traditional software development environments, organizations performing diagnostic record keeping in other domains or organizations performing problem-solving in various work domains).

Methodological contributions include exploring the limits of research into work practices of a decentralized, distributed community while relying on examination of artifacts (bug reports) and documentation on the associated organizational processes. Combining the application of grounded theory with the social informatics approach here demonstrates the ways in which these techniques are complementary.

Participant-observation, ethnography and interviews are well-established methods of conducting empirical studies in social informatics and in workplace studies. However,
these established methods also have important drawbacks and barriers to use. For example, participant-observation or ethnographic approaches depend upon co-location of the researchers and the population being studied for a long period of time—usually weeks or months of direct engagement. Interviews can be conducted across distance, but suffer from decontextualization and, if focused on past behavior, rely upon the interviewees’ memories. In-depth contextualized research, like ethnographic and participant-observer approaches, are usually limited in the number of subjects involved due to both the time needed to collect and prepare the data and the potential expenses and barriers to co-locating researchers with subjects.

This study centers on rich, primarily textual information contained in publicly available databases (bug report repositories). The community has created this information over several years, creating hundreds of thousands of different bug reports. F/OSS development communities are distributed, virtual, networked organizations that could be studied only with great difficulty using an ethnographic or participant-observer approach. Methodological innovation is required here because there is no specific physical location where a researcher could go to study the community.

This project makes theoretical contributions in three areas. First, it improves our understanding of the ways in which distributed work “gets done” through the empirically grounded analysis. It integrates existing theory of human information behavior; negotiated order; negotiation; coordination; trajectories and sensemaking with the social informatics approach to information studies in order to map the work of a complex, heterogeneous distributed community. Second, it makes an explicit bridge between research traditions in Library and Information Science, social informatics and human information behavior on one hand and research traditions like computer-supported cooperative work (CSCW) and distributed collective practices (DCP) on the other. This bridge is made by applying established theories of human information behavior, from Library and Information Science and related fields, to the study of distributed work. This project extends and generalizes work in human information behavior to include a wider variety of kinds of work by a widely divergent community of people, the users and
producers of a consumer-oriented application program. The information behavior of individuals and small groups in the community is linked to a larger set of distributed information practices. Finally, this project makes a significant addition to the existing base of theory related to software development and maintenance practices. Specifically, this project uses empirical data to explain how information is marshaled, negotiated, created and used in distributed SWPM work. The relationships between these phenomena will be used to support the extension of existing and the development of new theory about how software problems are identified, represented, managed and sometimes resolved in a distributed, communal, negotiated process.

Direct, practical contributions include applying the information drawn from the grounded descriptions of the work to the development of tools to improve the software problem management process. The empirical findings support the idea that development of computationally based subsystems, like a duplicate bug report detector, could provide important productivity improvements to a large distributed SWPM community. The findings also reveal that there are significant numbers and varieties of other types of relationships between bug reports, and that these networks of inter-bug report relationships create large and complex bug report networks that could better be utilized by the community through the application of computational techniques for summarization, quantification and visualization. Finally, the qualitative coding process itself can support the development, testing, and tuning of repository mining applications that analyze bug report text in support of either continued research or practical tool development to improve SWPM practice.
CHAPTER TWO: LITERATURE REVIEW

Introduction

This chapter reviews literature from a variety of domains in order to provide the reader with a basis for understanding the remainder of this dissertation. This project examines a complex socio-technical setting in a broad way, necessitating the integration of information from several disciplines and research traditions. There are three major subsections in this chapter. The first section focuses on the domain, software problem management, which itself is a part of software engineering practice. This section begins by identifying the reasons why software quality is important for economic reasons and in order to provide software that is usable, dependable and safe. Examples of how software quality has been studied in the past are reviewed, providing the context within which the current study has been executed. Existing research into the relationship between software development process and software quality is presented. Then, recent commentary and research into the ways in which free / open source software (F/OSS) development is perceived to be a significantly different and novel process is presented, including the argument made by F/OSS advocates that the F/OSS process can produce software of higher quality than traditional software development processes. Current research into F/OSS development processes is presented and critiqued, and gaps in the research record are identified.

The second section surveys a number of existing possible approaches to the analysis information, activity and social order in distributed work. Special attention is given to the large body literature often referred to as information needs and uses and, increasingly, human information behavior. Several other approaches that have been applied to the naturalistic study of complex socio-technical settings are reviewed, including Anselm Strauss’ theories of negotiation, negotiated order and trajectory; Weick’s sensemaking theory; several theories of coordination and cooperation; situated action theory; activity theory, distributed cognition; organization type and information use; and collective mind.
The final section describes and justifies the analytic process applied in the current project which is a synthesis of several of the possible approaches identified in the second section. The analytic approach used in this project combines approaches from multiple traditions: LIS’ human information behavior approaches and a variety of approaches applied in fields such as computer-supported cooperative work (CSCW), participatory design, cognitive science and organizational studies. The combination of these approaches demonstrates the utility of building bridges between the work of research communities working on similar problems with different tools and points of view.

Software Quality and Free / Open Source Software Development

The Importance of Software Quality

Software quality is an important concern for those who produce software, for software consumers and users and for researchers in software engineering and other related fields. As Abbot (1990) noted, the two reasons software quality is important are, first, the operational consequences of software problems and, second, the costs to identify and fix the problems themselves (p. 36-37). Low quality software releases can cause commercial software companies to lose market share (Court, 1998) or can lead to lawsuits against the software producer (Gilbert, 2002). Installation of low quality software can increase costs or cause purchasing organizations to lose revenue (Gilbert, 2002). Finally, as detailed in Chapter One, low quality software can have significant negative impacts on human beings and society (Neumann, 1995; Leveson & Turner, 1993; Gage & McCormick, 2004).

Software Quality Research

Practitioners and researchers have long recognized the software quality problem and sought to define it, understand what factors cause software quality to improve or decline, develop effective methods of measuring quality in real, large-scale software
development projects, and develop software development and software maintenance methods that improve software quality.

The goals and challenges of research in software quality are to:

[E]nable the software industry to deploy software products and services that are safe, dependable, and usable within an economic framework allowing companies to compete effectively…. Software quality concerns are quite broad, including, for example, correctness, robustness, readability, and evolvability. There is no single monolithic measure of software quality…. (Osterweil, 1996) (p.738)

The well-known and large body of work, by both practitioners and researchers, on software development methodology is targeted at finding better means of producing “safe, dependable, and usable” software on schedule and, importantly, within budget. Many of the approaches to improving software quality imply increasing the costs of production: adding specification reviews; adding code reviews; increasing resources devoted to testing; etc. Osterweil states (without explicit attribution to his sources) that “A number of studies have suggested that 50-60% of the effort involved in producing large software systems is devoted to quality assessment activities such as testing” (Osterweil, 1996) (p.738).

Much of the research on software quality has been performed in the software engineering community. Most of this work focuses on identifying, operationalizing, and applying quantitative metrics to enable use of statistical process controls. Osterweil (1996) reviews the existing software quality research and identifies several broad classes of informal and formal approaches. The informal approaches include dynamic testing, static analysis, and symbolic execution. Formal methods rely upon use of approaches including specification languages, model checking, and theorem proving.

Schneidewind (1999) examined the reliability of and stability of software maintenance processes and developed an integrated model to determine the how maintenance actions and testing affect software reliability. The goal of this work was to identify metrics to indicate whether a particular maintenance process was more or less stable at different points in time. This model relies upon quantitative metrics like mean
time to failure, total failures per KLOC (thousand lines of code), and total test time per KLOC.

A study of software maintenance processes at NASA sought to identify the “maintenance environment and processes in order to collect suitable and cost-effective data” in an effort to improve software maintenance projects (Briand, Basili, Kim & Squier, 1994). This study takes a wider view and takes factors such as organizational entities, documentation, inter-personal and inter-organizational communication, process phases, activities, and problem and error analysis into account in developing a model for how to characterize software maintenance organizations and processes. The authors argue that a holistic qualitative analysis is an important precursor to the development of quantitative models and management controls for software maintenance organizations.

Fenton and Neil (1999) provide a critical review of software defect prediction models including work on developing software complexity and size metrics, studies of testing processes, studies of software development methodologies, and multivariate statistical approaches. They propose application of Bayesian Belief Networks to the problem of defect prediction. They also argue that “we need compelling and sophisticated theories that have the power to explain the empirical observations (p.687).”

In an examination of the effectiveness of software development technical reviews (e.g., code reviews, code inspections, etc.), Sauer, Jeffery, Land and Yetton (2000) propose a research program based upon existing behavioral research on group performance. Application of group performance theories, developed during several decades of research on group performance in domains other than software engineering, to one group of techniques regarded as important in improving and maintaining software quality, allows the authors to develop testable propositions in the software engineering domain. They cite existing studies that have studied similar phenomena within software engineering or other domains as provisional support for their suggested research program. Their early assertions include that individual expertise is the dominant determinant of group performance; addition of people to groups does not result in commensurate performance increases; and that task training can have a positive impact
on group performance. This work is also notable and different in that it applies research from outside software engineering to software quality research.

Other efforts attempt to provide rapid and timely feedback to developers as defects are found and classified. These efforts attempt to help development projects that are in process identify emerging bug patterns as they occur so that immediate remedial action can be taken. The work associated with orthogonal defect classification (Chillarege et al., 1992) is in this tradition.

A smaller number of studies combine qualitative and quantitative analysis of software quality. Researchers interested in improving software reliability models used anecdotal information about how fixing bugs can introduce new bugs into a software system to challenge an assumption common to many accepted software reliability growth models. The assumption was that bugs in software can be “perfectly” removed when detected: that is, a bug, once detected and removed, will never result in the introduction of new bugs (Ohba & Chou, 1989). The reality that debugging frequently introduces new bugs is supported also by experiments of individual debugging performance. In one such experiment comparing the debugging performance of expert and novice programmers, novices were likely (more than 20% of the time) to introduce additional, undetected bugs while debugging to correct another bug (Gugerty & Olson, 1986).

Organizational Structure, Software Process, and Quality

Other research has examined the relationship between organizational structure and software quality. Perhaps the earliest expression of the relationship to organization, process, and software form and quality was made in 1968. This has come to be known as Conway’s Law: “… organizations which design systems … are constrained to produce systems which are copies of the communication structures of these organizations. (Conway, 1968) (p.31)” Writing about how software designs should be modularized, Parnas noted that a module, or a portion of the design, reflects a division of labor as well as a logically defined “subprogram.” (Parnas, 1972, p.1054).
Brooks’ Law states that adding staff to a late project will make it even later because of the negative effects of newcomers’ learning curves and increased costs of communication and coordination (Brooks, 1995). Cockburn (1996) presents design patterns that illustrate the impact social factors can have on software architecture decisions. For example, Cockburn writes that in a project with many novice and few expert designers, the work can be organized such that the experts design the more general parts that the novices can then specialize (p. 43-44). In this example, the available resources are allocated to the portions of the design with which they best fit. More recent work on F/OSS development projects (e.g., Tuomi, 2003) also expresses the importance of the relationship between aspects of the organization and its social forms and the structure of the software.

Based on interviews with participants of nineteen large software development projects, Krasner, Curtis and Iscoe (1987) identify and analyze a variety of interpersonal and inter-organizational communications problems and how communication affects system quality. Their findings provide confirmation of research from the 1960s that indicated that programmers spend more time on communication than they do on direct programming tasks. The nineteen projects studied were all organized as traditional rather than OSS development projects, so much of their work relates communication activity to the hierarchical, bureaucratic organizational model typical in traditional software development settings. Communication breakdowns identified include: no communications between groups who need to communicate; miscommunication due to unstated, unshared assumptions and contextual information; conflicting information from multiple sources; breakdowns due to “project dynamics” (“changes in people, goals, technology” p. 54). One specific kind of project dynamics problem – an example of Conway’s Law – occurred because “the structure of communications among project personnel conflicted with the most efficient decomposition of the system” (p.55). They quote one engineer: “In fact, it was more important to minimize the interfaces between

---

4 Later research into Brooks’ Law found that “Adding more people to a late project always makes it more costly, but it does not always cause it to be completed later.” So, there are always negative effects. (Abdel-Hamid & Madnick, 1991).
systems engineers than it was to make the system logical from the viewpoint of the user” (p.55). The factors that can contribute to or help minimize communications breakdowns were identified as communication skills; incentive systems; shared representational formats; conditions of rapid change; local jargon; breakdowns in information capture; and cultural mores for individual behavior.

Several examples of coordination and communications problems are described in a paper on a globally-distributed commercial software development project Herbsleb & Grinter (1999). The project relied upon project plans, software specifications and standard processes but encountered enormous problems, especially when the products of different groups were being integrated into a complete system. Herbsleb & Grinter found that a lack of readily available and established informal communications channels exacerbated technical problems encountered during integration. Citing one example from their study, team members at one site would, in an effort to complete their work as quickly and efficiently as possible, make adjustments and assumptions that affected the behavior of software interfaces but not formally document them. Team members at a remote site would, however, rely upon the outdated documentation as they implemented their part of the system, leading to difficult and subtle problems during later integration testing. Ineffective informal communications exacerbated the situation because (1) people in the two groups didn’t know each at the start of the project, (2) were from different national cultures and (3) spoke different languages. These factors were barriers to effective informal communication by telephone or e-mail. Informal communication and project effectiveness improved as members moved between the sites to work directly with their counterparts, establishing stronger social relationships. The people who spent time working at the other site became liaisons who were able to support/enable/foster direct contact between other people at both sites, simply by having a better understanding of who knew what at the remote site. In one example, Herbsleb & Grinter (1999) describe how geographically distributed programming groups built numerous assumptions into their software interface specifications being created by other groups that allowed mistakes to be built into the design and not be revealed until much later in
the project when the products from the distributed groups were being integrated into the final, unified software product. Geographic team distribution inhibited and limited the opportunities for informal contact that may have revealed these faulty assumptions earlier when their correction would be far less costly.

Software methodologies have historically been seen as a way to both manage projects (especially in terms of delivering software on time and within budget) and to manage quality. Even early “waterfall” processes were intended to provide a repeatable, generalized template that could be applied to any software development project. The Capability Maturity Model (CMM), developed at Carnegie-Mellon’s Software Engineering Institute, was intended to provide organizations with a roadmap that would guide the development of effective practices and processes and ultimately improve project manageability and software quality (Herbsleb et. al., 1997). Achieving higher levels within the CMM depends upon creation, adoption, and documentation of project management processes; managing requirements; managing quality assurance efforts; conducting peer reviews of processes and outcomes; implementing quantitative process measurement systems; and adopting defect prevention, process change, and technology change processes. The CMM, according to results reported in Herbsleb et. al., is most effective in medium to large software development organizations embedded within traditional organizations. Of more general importance is the finding that organizational and process factors do have a direct influence on software quality (Diaz & Sligo, 1997; Herbsleb et. al., 1997).

In an interview-based study of software project managers in two different cultures, the United States and Japan, Duvall (1995) takes account of the context within which software development projects take place and identifies and analyzes the problems software project managers encounter (e.g., requirements elicitation, specification and management), the situations within which these problems arise, the effects of these problems and what the managers do to address these problems. Duvall identifies contextual factors (which she calls problem sources) that contribute to the problems: society, the organization, the project, people, process and product (p.115). She also
identifies how these contextual factors differ in importance between the U.S. and Japan. For example, she notes several differences between Japan and the U.S. in terms of programmer competence, identifying training in school and on the job as an important factor. Managers in the U.S. expect college graduates to be productive immediately after they are hired while in Japan, there is several month training period after which young employees are expected to still need careful mentoring. Duvall notes that college education in the U.S. is seen as more rigorous than in Japan and programmers in the U.S. are seen as having a higher level of self-initiative than their counterparts in Japan. It is also more common for programmers in the U.S. to pursue another degree while employed. Duvall’s study is important as an example of a holistic, social informatics-informed study of software development work that acknowledges the “multidimensional social and technical nature of software development and the effect of restrictions and constraints” (p.122), often rooted in contextual factors, on an organization’s software development practices and products.

**The New Silver Bullet: The Free / Open Source Software Process**

Practitioners and researchers have been seeking a unified, universally applicable set of methods and tools to make software development efficient, manageable and economical. Brooks (1995) refers to this as the “silver bullet” that can slay the monsters that derail development projects. OSS has sometimes been promoted as the long-sought ideal development model.

F/OSS is primarily defined by the terms of the license terms under which it is distributed. The “free” in free software relates primarily to the limited constraints imposed by the free software license on the software’s users rather than the price of the software to the user. The most well-known free software license is the GNU General Public License (GNU GPL) (DiBona, Ockman, & Stone, 1999). One of the key provisions of the license is that, as a user, you have the right to make modifications to the software, but your modifications must also be distributed under the terms of the GNU
General Public License. Another key provision is that software protected by the GNU GPL cannot be incorporated into other systems covered by a license that provides more restricted rights than those provided by the GNU GPL. The Open Software License (Open Source Initiative, 2004b) provides similar protections to programmers and users. For software to be certified as open source compliant, it must meet the criteria defined by the Open Source Definition (Open Source Initiative, 2004c). For a comparative discussion of the sometimes-subtle differences between these and other license variants associated with F/OSS systems, see Perens (1999).

The F/OSS process has been compared to the scientific method (DiBona, Ockman, & Stone, 1999) by some commentators. The complete availability of the source code is seen as similar to researchers’ publication of data and methods; the distributed, informal organizations and processes that support F/OSS development is similar to the invisible colleges within which science is organized; the primacy of meritocracy: like citations to important scientific papers, the code deemed the “best” after review by other programmers is the code that is included in the F/OSS system or application and used by larger constituencies. Being recognized as an important contributor to a F/OSS project builds the reputation of the contributing programmer in ways similar to how a scientist’s reputation is built by writing oft-cited papers or books. The Open Source Initiative states “Open source promotes software reliability and quality by supporting independent peer review and rapid evolution of source code” (Open Source Initiative, 2004a).

The genesis of F/OSS projects is usually described as organic and unplanned. McKusick (1999) describes how one programmer, Keith Bostic, “pioneered the technique of doing a mass net-based development effort” (p.42). The eventual result was the creation of a free, open source version of Unix (BSD, the Berkeley Software Distribution) in 1991. Linux began as one student programmer’s project to create an operating system with a Unix-compatible interface for the Intel 80386 processor. In general, most F/OSS projects begin when a developer encounters a situation where his or her tools are inadequate for the job at hand. Raymond states this as lesson one in the essay “The Cathedral and the Bazaar” (Raymond, 1999) (p.32).
Other models of project genesis have also emerged, including several “hybrids” that combine elements of the traditional F/OSS process with participation by commercial firms in various ways. In January, 1998, Netscape announced that it would “give away the source code for the next generation of its Communicator suite. (Hamerly, Paquin & Walton, 1999) (p.197).” This surprising announcement, made by a large, heretofore successful commercial company, led quickly, in less than three months, to the establishment of Mozilla.org, the organization that would go on to create the F/OSS Web browser called Mozilla. Some of the critical activities that had to be completed prior to the release of the source code were removal of proprietary, third-party software (Java, for example) and the selection or development of an appropriate license. The creation of Mozilla.org also occurred prior to the release of the source code, which also was different and more formalized than the organic organizational evolution typical of F/OSS projects (Hamerly, Paquin & Walton, 1999).

Differences between traditional and F/OSS engineering practices have also been identified. Among these are that F/OSS projects typically place less emphasis on formal documentation (requirements specifications; system-level design documents; detailed design documents; quality assurance and test plans) (Scacchi, 2002). On the other hand, F/OSS projects benefit from a relatively large body of motivated and technically adept testers who may submit software patches when they identify software problems. This user community, which is encouraged to identify and fix problems as well as improve and extend the design and implementation, also provides a relatively large support group. Not all F/OSS projects match the generalizations made above. The Mozilla project, for example, has an extremely sophisticated set of processes and tools for ensuring quality and support of testing and the management of software problems. Most F/OSS projects are small enough in scale that design information is held in the heads of one or a few programmers5 (Vixie, 1999).

5 The code itself, of course, is the most accurate representation of the system design of any software whether developed using open source or traditional approaches.
Many commentators also claim that F/OSS is of higher quality than software produced by traditional software development methods. Raymond makes a distinction in project styles within the F/OSS community between “cathedral” and “bazaar” style projects. Cathedral style projects are similar in some ways to traditional commercial software development projects – these projects are likely to have infrequent and carefully controlled releases, with intense and centrally controlled bug fixing and new feature development between releases. Bazaar style projects, like Linux and Mozilla, are likely to have frequent releases at intervals of a few days and little centralized control over the evolution of the design. Another oft-quoted lesson from Raymond is lesson eight, also referred to as “Linus’ Law”: “Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone, or less formally, ‘Given enough eyeballs, all bugs are shallow.’” (Raymond, 1999, p. 41). An optimistic point-of-view is taken in bazaar-style projects, where bugs are considered to be, generally, no big deal, and, when they are identified, they will be fixed quickly by someone within the community.

**Free / Open Source Software Research**

Free / open source software (F/OSS) research is an emerging research area. The existing body of research is small; many of the studies are general and exploratory in nature; few are truly empirical; many are small-scale; few address the issues of SWPM directly.

Some of the research is quite broad in nature (e.g., Benkler, 2002; Tuomi, 2003) and examines F/OSS software projects as exemplars of phenomena of interest to a more general public (economists, policy makers) and not just to software engineers, software firms and programmers.

The processes and tools used in the Mozilla project are described and discussed in Reis & Mattos Fortes (2002). While, strictly speaking, not a research paper, it provides a comprehensive introduction into the methods employed by one F/OSS project to
manage the software development process. The authors acknowledge the rich data available in some of the tools used to support the development effort, particularly the code management systems and the bug report repository, Bugzilla, noting several interesting research projects that could be carried out based upon that data. This paper also points to interesting conventions and behavior among Mozilla community members. For example, overall system design changes occur “continuously,” which is a typical design approach in F/OSS projects. In some cases, requirements specification and design are driven by high level architectural objectives (e.g., Mozilla will be a standards-compliant browser – see page 8). More often, new requirements are first brought up in project newsgroups and discussed; when some consensus is achieved, the requirement is expressed as a Bugzilla bug report. If enthusiasm or support for the idea flags, a bug report is never created. In other cases, the initial requirements statement will be expressed in a bug report.

Benkler (2002) addresses the skepticism often expressed by observers about F/OSS projects and their ability to survive and thrive. Questions about how these projects are organized, how the projects maintain momentum and quality, and how the participants, usually volunteers, are motivated to participate are among those often raised. F/OSS models do not make sense to many people when they examine them using the two traditional lenses of markets and firms.

Benkler describes what he calls peer production of information and culture and makes the claim that peer production is an emerging, third organizational form. Peer production occurs frequently in projects and systems on the Internet and is applied to information creation, information accreditation (vetting), and information distribution. Benkler’s analysis is based upon examination of a broad variety of Internet-based systems and projects which include F/OSS projects. Referring to F/OSS projects in particular, Benkler notes that peer production is used to ensure software quality by allowing submission, by multiple people, of redundant (redundant in the sense that it is superficially functionally equivalent) code to a particular project and thus providing project coordinators a means of averaging out similar contributions or preventing low-
quality submissions from actually entering the code. Benkler argues that peer production is increasing in visibility and importance because of declining costs associated with information production and increased communications capability due to the Internet. These conditions in turn lower the costs of coordination of work and “increases the importance of the factor at which peer production has a relative advantage – identifying the best available human capital in highly refined increments and allocating it to projects” (p. 444). Peer production applies to software problem management in F/OSS projects, and will be discussed in more detail in subsequent sections of this report.

Tuomi (2003) also looks at the F/OSS development process from a broad perspective, particularly from the point of view of innovation research. Tuomi examines how the development of Linux illustrates the tensions between fostering innovation and maintaining some control over system function, maintainability and the social relations that make coordination of Linux’s development possible.

Software Problem Management Research

Much of the literature on software problem management (SWPM) consists of problem solving techniques, tricks, tool design and heuristics applied at the individual level. Early studies of software development include reference to this kind of material (Brooks, 1995 (originally published in 1975); Weinberg, 1971). Typical recommendations in these prescriptive approaches include careful reading and understanding the code, controlled experimentation with test data, formulating and testing hypotheses, “divide and conquer” to localize the source of the bug, check for recent changes, stabilization of the error condition to make it reproducible, talk the problem through with someone else, check for common problems (variable initialization; loop conditions; etc.), perform a code review (McConnell, 1993) pp. 623-649).

Literature examining individual debugging performance is also common. One experiment compared the debugging performance of expert and novice programmers while debugging simple programs in two languages. This study concluded that experts
debug in less time, are more likely to solve the problem, are less likely to introduce additional bugs, and are more likely to identify the correct hypothesis to test on the first try than novice programmers (Gugerty & Olson 1986). Similar results have been found in other similar experiments (Gould, 1975).

Little research has looked at debugging as a human and social process. Some early software development works hint at aspects of the social process. Brooks (1995) for example, talks about the importance of communication and control within an organization (through documentation and process) as he discusses “system debugging” in Chapter 13 of *The Mythical Man-Month*.

In a study of the information gathering strategies of maintenance programmers at two organizations, a combination of formal documents (requirements and design documents; test plans and reports; source code; comments embedded in the source code; user documentation), empirical experience (using the software) and informal methods (talking to the original or other experienced developers of the system) were used as means to gain information about the system (Seaman, 2002). Bug reporting repositories were not cited in this study as an information source, but “lessons learned reports” were valuable in helping maintenance programmers avoid making previously committed errors. There was no consensus on the utility of lessons-learned reports: respondents felt they were poorly organized, poorly maintained and suffered from lack of attention and resources dedicated to their development.

One example of the kind of analysis that forms the core of this project is described in a paper comparing analytic frameworks to the study of software development work. The paper describes an incident of a programmer and his organization responding to a user’s bug report as an illustrative example applying multiple analytic frameworks. The incident is described as occurring within a specific social context where the roles of many people (the reporter, the help desk, the programmer and the programmer’s manager) are part of the story. The incident is focused on the contexts, processes and social order of the event rather than in the steps taken by the programmer to fix the bug (Spinuzzi, 2001).
Berlin and Jeffries (1992) examined cooperative problem solving between programming “apprentices and experts” (p.130) in a software development organization. The focus in this study is the primary face-to-face interactions between the experts and apprentices as the apprentices seek to learn both a new computer language and how to become productive members of their software development teams.

A study performed at Bell Labs elicited the expertise of expert debuggers in order to support the development of a course for developers on software debugging skills. This information, presented in a series of papers, represents a rare example of empirical, rather than anecdotal, investigation into how individuals debug. In one of these papers, the authors emphasize their surprise at learning how important a component “social and psychological expertise” is in debugging expertise (Riedel, et. al., 1991). They attribute this to the social aspects of the development of large software systems: systems are too complex to be created by or even understood by individuals. When bugs occur, the debugger must interact with others, including other experts (other developers or designers) and possible non-experts (non-technical users who may have identified or reported the problem in the field). Describing the course content derived from the empirical study of expert debuggers, the authors describe a model of the debugging process presented to students. In this model, the experts “…first clarify the bug symptoms and refresh their memory of the domain… then cycle between testing hypotheses… and incorporating… their newfound knowledge into their existing understanding or representation of the system. When their knowledge of the system clearly explains the symptoms, the programmer has found the error (Freeman, et. al., 1991).” (p. 276)

Empirical studies of software problem management from a social or organizational perspective are rare. Carstensen, Sørensen & Tuikka (1995) provide a description of the normative SWPM process used in a manufacturing firm. The system they described was a paper-based bug report repository in a setting where all of the developers and software designers were collocated. The purpose of their paper was to use their case study of software testing and bug report management as a basis for developing
improved methods for modeling cooperation with the goal of creating formal specifications for use in the development of CSCW systems. Their description and analysis of software problem management is somewhat superficial because they are using bug reports, a resource-planning spreadsheet, and a procedure for controlling system configuration, using these artifacts and related processes as an exemplar for a general approach to supporting articulation work with computer-based information systems.

Schmidt & Simone (1996) refer to and make extensive use of Carstensen, Sørensen & Tuikka’s analysis of the bug report form as one of four types of artifacts (the others are checklists, kanban (cards), and standard records for office accounting) upon which they base their theory of coordination mechanisms.

Crowston (1997) uses the software change process of a mini-computer manufacturer, including the company’s software problem management process, as the empirical basis for the development of coordination theory. Crowston’s detailed description of the software problem management process is the most complete and carefully drawn description of the process documented in the research literature to date. He intends coordination theory to be used to analyze business processes in order to perform business process re-engineering. Kajko-Mattsson (1999) also provides a detailed description of the SWPM process used in one company but offers little additional insight.

Tuomi (2003) presents a description and analysis of the quality control process used in Linux development. Tuomi starts with Raymond’s (1999) explication of the social processes at play in F/OSS development projects and Raymond’s characterization of “Linus’s Law” (quoted above). Tuomi explains how the software quality process plays out in Linux, a large F/OSS development project where a few hundred core and several thousand additional co-developers are involved in the software quality and software problem management processes. Unlike traditional software development projects, the source code is available to all of these thousands of co-developers, making it possible for any one of them to either report, characterize or fix problems, thereby increasing the probability that a problem will be addressed quickly with a well-designed, high-quality fix.
Within F/OSS development communities, the relatively large number of co-developers is also the group who performs beta-testing and the majority of bug reporting and repair.

The software problem management process described in Linux documentation is virtually the same as the process described in the introduction, above. Tuomi notes that the normative process given in the documentation is simplified and does not describe the social processes, tools, and other resources required to effectively manage software problems.

The normative process provided in the introduction, above, is based upon the few previous research projects described above (Carstensen, Sorensen & Tuikka 1995; Schmidt & Simone 1996; Crowston, 1997; Kajko-Mattsson, 1999; Tuomi, 2003), my own experiences using a variety of problem management systems in several different settings during my professional career and documentation associated with F/OSS bug report repositories. The normative model, including a number of changes and additions to it, identified as a result of the analysis done in this research project, is re-presented in Chapter Nine.

**Approaches to the Study of Information Practices in Distributed Work**

This section reviews a number of possible analytical approaches that are applicable to the study of complex socio-technical settings such as software problem management.

**Information Needs, Uses, Behavior, and Practices**

The information needs, uses and behavior literature represents one of the central research areas in Library and Information Science (LIS). Conceptualizations of the terms information needs, information uses and human information behavior have varied. The term information practices has also begun to be used in the research literature as well. This section begins with a brief discussion of the terminology, followed by a review of key research, concentrating on work published since 1980. Reviews of the existing literature in this area
(including key reviews such as Dervin & Nilan (1986); Hewins (1990); Leckie, Pettigrew, & Sylvain (1996); and Pettigrew, Fidel & Bruce (2001)) have noted at least four important trends: from a focus on individuals or homogenous groups (e.g., aerospace engineers) toward heterogeneous groups engaged in collective information practices; from the study of “elites” (often considered as a homogeneous group) in constrained settings toward increased interest in the study of individuals with a wide variety of demographic backgrounds, professional or casual interests and in “everyday life situations”; from methodological foci on either experimental methods aimed at revealing the cognitive processes of individuals or the quantitative description of typical information behaviors of a group toward naturalistic inquiry into human information behaviors in realistic contexts; and from system-centered orientation to user-centered orientation in specific and often complex contexts (including a concomitant shift from quantitative methodological approaches toward qualitative and multi-method approaches).

The terms information needs and information uses lack precise definitions in the existing research literature. These terms are useful not as precise labels but only as very general markers for research having some regard for information users, information seeking and the uses to which information is put by users. Wilson (1981), followed by Dervin and Nilan (1986), argue that much of the early research conducted under this rubric had limited usefulness based upon its vague assumptions and definitions. One criticism is that this terminology (and the underlying research) is insufficiently precise to either help develop useful general theory, provide insights that can lead to improvements in the practices of information institutions (like libraries) or provide empirically justified approaches for improving the design of information systems.

Wilson (1981) suggests that a focus on the contextualized information behavior of individuals will lead to more holistic conceptualization of the information user, shifting “the focus of research from an examination of the information sources and systems used by the information-seeker to an exploration of the role of information in the user’s everyday life in his work organization or social setting” (p.10). Wilson also emphasizes the criticality of context in such studies, noting that
There can be little use … in a national survey of the ‘information needs’ of any group (chemists, botanists, economists etc.) if members of these groups are undertaking widely differing kinds of tasks in totally different organizations with varying levels of information provision (Wilson, 1981). (p.11)

Wilson later defined human information behavior as

… the totality of human behavior in relation to sources and channels of information, including both active and passive information seeking, and information use. Thus, it includes face-to-face communication with others, as well as the passive reception of information as in, for example, watching TV advertisements, without any intention to act on the information (Wilson, 2000). (p. 49)

Pettigrew et. al. (2001), in light of Wilson’s work, defined human information behavior as “how people need, seek, give, and use information in different contexts, including the workplace and everyday living” (p.44).

Subsumed within Wilson’s definition of human information behavior are more specific activities such as information seeking behavior, “the purposive seeking for information as a consequence of a need to satisfy some goal;” information searching behavior, “the ‘micro-level’ of behavior employed by the searcher in interacting with information systems of all kinds;” and information use behavior, “the physical and mental acts involved in incorporating the information found into the person’s existing knowledge base (Wilson, 2000)” (pp. 49-50).

Collective and Constructive Information Practices

Wilson’s conceptions are focused on analysis of the individual user of information, information systems and information services. McKenzie (2003) builds on Wilson’s notion of human information behavior by defining information practices as encompassing “the entire range of elements present in” accounts of information seeking, searching and use including “how information comes or is given through the initiative or actions of another agent” (p.25). Of particular note are that Wilson and McKenzie’s
(2003, 2004) conceptions of human information behavior / practices emphasize information use. McKenzie in particular, and Wilson to a lesser degree, point out the active creation or construction of information as part of the information practices found in a particular setting. McKenzie makes a crucial insight, recognizing how individuals work both face to face and at distance (that is, separated by time and / or space) to collectively seek, acquire, give and construct information.

The term information convergence has been used to describe complex processes that lead to more-or-less complete integration between an individual and multiple kinds of information and channels of information communication (Star, Bowker & Neumann, 2003). One research project revealed differences in information convergence experienced by users of a digital library of scientific journal articles who were at different stages in their careers: faculty members have well-established social connections into their fields that make it likely that information seeks them (through their local and remote colleagues; as reviewers for conferences, journals and grant proposals), rather than their seeking information in the active way often described in human information behavior research. Graduate students, by contrast, have less well developed social infrastructure and thereby experience attenuated information convergence (Bishop et. al., 2000). McKenzie’s work with pregnant women (McKenzie, 2003) reveals similar patterns of information coming to the user within a larger set of information behaviors.

The constructionist approach is also applied to the study of digital library design to help illuminate the role such systems can and do play in the life of the communities that use them. Tuominen, Talja & Savolainen (2004), for example, see digital libraries as systems that support “knowledge production” (p.561), not just a means of storing information. They call for expansion of traditional LIS studies and design approaches to accommodate knowledge production, conflicting perspectives and debate. Their social constructivist approach “maintains that while the mind constructs reality in its relationship with the world, this mental process is significantly informed by influences from social relationships” (p.564). Tuominen, Talja & Savolainen base their approach in part on Dervin’s work, which views knowledge as a fluid product of communication
through “dialogue, contest and negotiation” (Dervin, 1994) (p.377). Dervin and Nilan (1986) noted in their summary of the emerging “alternative” research paradigm:

[The “alternative” paradigm posits information as something constructed by human beings. It sees users as beings who are constantly constructing, as beings who are free (within system constraints) to create from systems and situations whatever they choose. It focuses on how people construct sense, searching for universal dimensions of sense-making. It focuses on understanding information use in particular situations and is concerned with what leads up to and what follows intersections with systems. If it] focuses on the user (Dervin & Nilan, 1986) (p.16).

Studies of a Wide Range of People and Settings

Many studies have been done into the information needs, uses and behaviors of particular groups of people, usually professionals. Allen (1977) studied engineers in research and development organizations, pioneering the use of the critical-incident method to elicit contextual descriptions of information behavior from his subjects. Taylor (1991) bases his information use environment (IUE) model on data from earlier studies of the information needs and uses of engineers, legislators and doctors. Taylor acknowledges the importance of studying other types of information users, but excludes them from his analysis. Bishop et. al. (2000) conducted a multi-year, holistic and multi-method study of the information behaviors of science and engineering faculty, students and librarians in a research university. Hirsh and Dinkelacker (2004) employ a survey to ascertain the criteria for selection and use of particular information sources in the research and development laboratories of the recently merged Compaq Computer and Hewlett Packard. Information needs, uses and behavior studies of professionals has sometimes been critiqued for its focus on elite groups, like scientists, engineers, medical professionals, university faculty and students, etc. It has yielded important results, especially in regard to identifying and describing the complex nature of individual information behavior, introducing innovative research methods and providing several
important models and the beginnings of a theory of information needs, uses and behavior.

Investigation of the information practices of people in everyday life situations has recently become a more common object of study. Dervin (1992) summarizes several early studies she and her colleagues conducted on the information behavior of non-elites including the information needs of California citizens, information-seeking by blood donors, public library use and non-use by Hispanics, information-seeking and use by college students, and the health information needs of Southeast Asian refugees. One of the important innovations in Dervin’s research is its focus on the individual’s conception of their own situation / questions / problem / information gap instead of the information behavior of members of a group (e.g., engineers). Her focus on the individual is a response to one of her criticisms of much information needs, uses and behavior research and its over-emphasis on demographic variables (e.g., social / professional class, race, age) as a predictor of human information behavior. Dervin and Nilan (1986) note that much early information needs, uses and behavior research assumed “that if we know who people are, what groups they belong to, or what their activities and interests are, we know what their needs are” (p.12).

Savolainen (1995) studied “everyday life information seeking” (ELIS) by interviewing elementary and vocational-institution teachers and industrial workers, a sample that includes people affiliated with a profession as well as blue-collar workers. Bishop et. al. (2000), while focusing on faculty and student users of an early science and engineering full-text retrieval system, recognized the biases of studies focused on elites, and included a wider variety of people in their studies, including high school students and teachers, system developers, reference librarians, as well as faculty, undergraduate and graduate students in any discipline. Bishop et. al. (2003) takes a participatory action research approach to the integrated design, development and evaluation of a community-based health information system. They partner with community members from a traditionally marginalized group to identify information needs and prioritize both development objectives and evaluation criteria and to ultimately change the local social
conditions affecting the health, health care and health information practices of African-American women, health care providers and institutions. This innovative approach erases the traditional boundary between social science researcher and subject and expands the notion of information and expertise by giving community members the opportunity to identify and prioritize project goals, needs and tasks in their terms rather than being constrained by the more typical role of research objects / subjects.

Studies into the information practices of people in more prosaic, less privileged work settings, like offices, manufacturing, retail or other production settings also appear outside the LIS literature. Examples of research into these settings are more commonly found in other traditions such as participatory design (Blomberg, Suchman & Trigg, 1997) and CSCW (Schmidt & Simone, 1996; Sandusky, 2003). These studies, however, rarely link back to the LIS information needs and uses or human information behavior literature.

Fisher, Durrance & Hinton (2004) take an important step by applying the conceptualization of human information behavior (along with Fisher's (nee Pettigrew) concept of information grounds) to the evaluation of practice of information services provision in a study of the use of public library-based information services provided to immigrants in Queens, New York. An information grounds (Pettigrew, 1999) is “an environment temporarily created by the behaviour of people who have come together to perform a given task, but from which emerges a social atmosphere that fosters the spontaneous and serendipitous sharing of information” (p.811). Fisher, Durrance & Hinton (2004) identify seven “key concepts” in the theory of information grounds: (1) context rich; (2) temporal setting; (3) instrumental purpose; (4) social types; (5) social interaction; (6) informal and formal information sharing; (7) alternative forms of information use (p.756). Information grounds is particularly useful as a way to address the difficult issues of context when seeking to understand human information behavior. Pettigrew (1999) identified four distinct “contextual factors” in her study of human information behavior in temporary foot clinics and noted that while “each category of contextual factors affect the flow of HSI [human services information] at the clinic in
some respect, when considered collectively, these factors create a ‘grand context’ from which broader conceptual findings may be drawn” (p.809). The notion of an information grounds thus provides a way of addressing the complex interrelationships between different contexts, which are invariably present when studying human information behavior in natural settings.

What is somewhat less obvious is an implicit assumption that the participants in an information ground are physically co-located: this is the case in both Pettigrew (1999) and Fisher, Durrance & Hinton (2004). Physical co-location is indicated by portions of the description of information grounds: information grounds “are predicated on the presence of individuals” (p.756); the prediction that “information grounds might occur in such settings as hair salons, barber shops, quilting bees, playgrounds, tattoo parlors, metrobuses, foodbanks, etc.” (Fisher, Durrance & Hinton, 2004) (p.757). The implicit constraint of co-location, however, does not necessarily preclude the application of the theory of information grounds to other modes of social interaction (e.g., distributed work settings); the theory is relatively new and may not yet have been applied in other kinds of settings.

Contextualized Inquiry into Human Information Behavior

Wilson discussed the power of qualitative methods to develop concepts related to the study of the use of information systems. He calls for contextualized inquiry into human information behavior, noting that research into information-seeking should attempt to uncover the facts of the everyday life of the people studied, the ways in which these facts motivate their behavior, the meaning the information found has in their everyday lives and through this understanding be able to provide more useful information systems (Wilson, 1981) (p.11).

Dervin and Nilan (1986) emphasize the emergence of research approaches that, instead of relying on a mechanistic point of view that assumes the user is a “passive recipient of objective information” (p.13), research should focus on behavior, including
the ways in which users actively construct their information needs in situations and specific contexts and the strategies they use for directing their information behaviors, including their own subjective interpretations of information and subsequent use of information. They also point to the early dominance of studies examining the limited context of the users’ interaction with a particular system (e.g., card catalog, OPAC, search interface) and the emergence of approaches taking a holistic view of information needs, uses and behavior. Dervin’s own sense-making approach to the study of human information behavior, incorporating the micro-moment timeline method of interviewing, is an exemplar of how information behavior research can be done in “user terms” instead of “system terms” (Dervin, 1992).

Taylor (1991) argues that “information use environments” are the appropriate unit of analysis because information use environments encompass the “contexts within which … users make choices about what information is useful to them at particular times” (p.218). The information use environment is:

The set of those elements that (a) affect the flow and use of information messages into, within, and out of any definable entity; and (b) determine the criteria by which the value of information messages will be judged (Taylor, 1991) (p.218)

Hewins (1990) emphasizes the need to link an individual’s information need to that individual’s application of that information to solving a problem, making sense, etc. She also emphasizes the importance of considering context in such research because context, including the individual’s personal characteristics, can influence all aspects of information behavior (seeking, searching, use, etc.).

Research that takes a holistic, social informatics approach to examining information practices is becoming more common, too. Bishop et. al. (2000) takes an inclusive research approach to the study of digital library use and infrastructure, combining interviews, observations, transaction log analysis, ethnography, focus group interviews, usability tests and on-line surveys of a wide variety of potential and actual users of the digital library. This broad methodological approach makes it possible to
examine phenomena at a variety of levels of granularity, including particular interface and system problems (revealed by usability testing and transaction log analysis) to the effects of changing infrastructure (including rapid development of the World Wide Web as well as development of a particular local digital library) on the information behaviors of both system providers (developers and librarians) and users (faculty and students). Such studies often seek to identify the varieties, processes, contexts and social order that collectively constitute information practice in a particular setting.

Bishop et. al. (2003) takes an even more contextualized focus by taking as a primary project goal the achievement of “constructive social outcomes” and the development of increased capacity of individuals and organizations in the community of African-American women with which they are engaged. McKenzie’s recent work (2003, 2004) and the work of Fisher, Durrance & Hinton (2004) are other examples of research that makes a serious attempt to address issues of context.

User Centered Inquiry

Dervin and Nilan (1986) presented evidence in the research literature showing a paradigm shift away from research that “observe[s] users in terms of systems” and toward research that “observe[s] users in terms of users” (p.9). They note as evidence of the system orientation of this research that “respondents are usually offered a menu of options that originate in system worlds, not user worlds” (p.12).

Taylor (1991) distinguishes the “user and uses of information” approach from two other approaches, technological approaches and content-driven approaches. Technological approaches to usage research place the system at the center of interest and tend to focus narrowly on the interactions between users and the system. Content-driven approaches, in Taylor’s sense, focus on examining methods and systems of organizing information (e.g., classification schemes). In his view, the user centered approach includes the contextual information he calls the “information use environment” (p.218). He categorizes the information use environment (IUE) into six categories: sets of people (e.g.,
professions, special interest groups, special socioeconomic groups, etc.); typical problems of a set of people; setting (e.g., organization, domain, level of access to information, etc.); resolution of problems (e.g., determination of a fact vs. “satisficing”); nature of information (e.g., embodied in an object; perceived; etc.); and processes of decision making (e.g., algorithmic and predictable; negotiated and contingent, etc.) (pp. 221-233; 247-148).

Dervin and Nilan’s (1986) emphasize the need for and note increasing evidence in the research literature of the development of “rational frameworks and systematic bases for methods and definitions” (p.146) in user-centered information needs and uses research. Subsequent reviews of the information needs and uses literature, such as Hewins (1990), confirm Dervin and Nilan’s assertion that user-centered research is becoming a dominant paradigm. Hewins does, however, see improved “dynamic, adaptive” (p.165) systems design as a major goal of user-oriented information behavior research, indicating her interest in the application or user-centered research findings to systems development.

Negotiated Order

Day and Day (1977) (as quoted in Strauss, 1978, p.260), provide this summary of negotiated order:

In the case of negotiated order theory, the individuals in organizations play an active, self-conscious role in the shaping of the social order. Their day-to-day interactions, agreements, temporary refusals, and changing definitions of the situations at hand are of paramount importance. Closely correlated is the perspective’s view of social reality … the negotiated order theory downplays the notions of organizations as fixed, rather rigid systems which are highly constrained by strict rules, regulations, goals and hierarchical chains of command. Instead, it emphasizes the fluid, continuously emerging qualities of the organization, the changing web of interactions woven among its members, and it suggests that order is something at which the members of the organization must constantly work. Consequently, conflict and change are just as much a part of organizational life as consensus and stability. Organizations are thus viewed as complex and highly fragile social constructions of reality which are subject to the numerous temporal, spatial, and situational events occurring both internally and externally. (Day & Day, 1977) (p.132); cited in Strauss (1978), p. 260)
Strauss’ (1978) states that negotiated order refers to the situation that holds whenever people use negotiation, one process by which people work together to “get things done, to reach common as well as private goals” (p.121). Thomas (1984) uses the concept negotiated order to analyze social interactions in prisons, which are typically thought of as institutions where negotiation would never occur. SWPM is one kind of work in which some aspects are predictable and routine (like the normative process the community or organization uses to handle a bug report), and some are unpredictable (the particular steps a programmer takes to repair a software problem). We will see that one of the basic social processes the community uses to deal with uncertainty is negotiation. Negotiations can be conducted on any topic, routine or non-routine. The non-routine situations are often more interesting because the bug reports reveal how the community adapts to uncommon or uncertain conditions. F/OSS development communities in particular are less hierarchical and less coercive than traditional software development organizations, which are usually embedded in traditional bureaucratic organizations. Negotiation, as a way to get things done, and a way to enact and maintain social order, is likely to be prevalent.

**Trajectories**

A trajectory is a path, progression, or line of development resembling a physical trajectory. The concept is used here in the sense developed by Strauss, Fagerhaugh, Suczek, and Wieder (1985) in their work on the social organization of medical work. In defining the concept, they first distinguish the course of an illness from the illness trajectory.

A distinction central to the analysis presented in this book is that drawn between a course of illness and an illness trajectory. The first term offers no problems to the reader since everyone has experienced an illness that did not merely appear but developed gradually over time, getting worse and then perhaps clearing up. To the knowledgeable medical, nursing, and technical staffs, each kind of illness has its more or less characteristic phases, with symptoms to match, and often only skilled intervention will reverse, halt, or at least slow down the progress of the disease.
Course of illness is, then, both a commonsense and professional term (Strauss et. al., 1985). (p.8)

The authors contrast the concept “course of illness” from “illness trajectory” in this way:

Trajectory is a term coined by the authors to refer not only to the physiological unfolding of a patent’s disease but to the total organization of work done over that course, plus the impact on those involved with that work and its organization. For different illnesses, the trajectory will involve different medical and nursing actions, different kinds of skills and other resources, a different parceling out of tasks among the workers (including, perhaps, kin and the patient), and involving quite different relationships – instrumental and expressive both – among the workers. (Strauss et. al., 1985). (p.8)

Finally, the authors speculate about the utility of the notion of trajectory in other settings:

Of course, all work – industrial, commercial, artistic, domestic – involves a sequence of expected tasks, sometimes routinized but sometimes subject to unexpected contingencies. It may be that trajectory fits the organization of those kinds of work also (Strauss et. al., 1985). (p.9)

The contents of a bug report – the comments, the attachments, and the bug activity data – represent a bug report’s trajectory from the time the bug report is created until it is set to a terminal state (e.g., ‘status = verified’). But the bug report also reveals information about how the work of managing software problems is aligned; who participates in that process; how the organization of the community affects and is affected by the information contained in the bug report.

The term trajectory is useful because the current project is concerned not only with the identification and correction of the “bugs” in the software, but also with developing an understanding of how software quality work is performed; how the work is organized; identifying the kinds of contingencies that occur and the methods by which they are handled; how different tools are applied; how the relationships between community members with different roles and skills are established and maintained, etc.
Strauss et. al. (1985) also point out that there are two “striking features of health work” that are not shared by all other kinds of work. First, “the unexpected and often difficult to control contingencies stemming not only from the illness itself, but also from a host of work and organizational sources” (p.9). Second, “it is ‘people work.’ The product being worked on, over, or through(!) is not inert, unless comatose or temporarily nonsentient” (p.9). Software problem management in this F/OSS development community is similar to medical work in terms of the first feature because SWPM is affected by contingencies as well: unexpected interactions in the system; sudden re-prioritization of work; people unpredictably leaving or joining the community; etc. Software problem management is certainly different from medical work in the second regard. Software and bug reports, the products upon which SWPM is done, are not living, sentient creatures; while there may be rare exceptions, software problems generally will not “get worse” if they are not worked on, unlike a patient with an infection.

Situated Action Theory

The situated action approach to the study of human behavior focuses on the relation between the actions of a person or persons in a setting, where setting is defined as “a relation between acting persons and the arenas in relation with which they act” (Lave, 1988, as cited in Nardi, 1996b). “An arena is a stable institutional framework” (Nardi, 1996b) within which action is situated.

Situated action research has a close focus on detailed analysis of human action. For example, Suchman (1987) explicates situated action in terms of a handful of interactions between people and a new photocopier. Lave (1988) examines improvisation in everyday problem solving in great detail.

While the situated action approach is relevant to this research, the focus of the current project is the analysis of bug reports, which are texts that summarize activity of multiple people in the management of a software problem. The detailed level of granularity typical of situated action research is not appropriate for the study of bug
reports done in this project. A situated action approach would be more relevant in a study whose design focused on direct real-time observation of or participation in the SWPM process.

What is relevant about the situated action approach to this project is the notion of the action of the participants in a setting is more or less improvisatory. Each bug report is unique, so the response of the F/OSS community is necessarily specific to the conditions and phenomena described in each bug report. Even if two bug reports are describing the same phenomena, they will not do so using the same descriptions, terms, etc. Consistent with situated action, what we see in SWPM management is situated responsiveness and improvisation on the part of the community to different, ambiguous situations represented in the bug reports. Activity in SWPM is structured on the fly and not prescribed by the normative process, written procedures, or reified in the BRR, the BR and the supporting protocols.

In the context of SWPM, the arena is the F/OSS development community’s coordination mechanism, the bug report repository, its bug reports, and the protocols (the normative SWPM process) that support SWPM. The setting is the ongoing, unfolding, co-constitutive relations between a particular bug report (an artifact with a certain set of affordances as defined by the bug report repository within which it exists) and the people involved with it (assignee, reporter, commentators, QAContact, etc.),

The bug report, bug report repository, and the protocols associated with it (in other words, the coordination mechanism) are resources for action (Suchman, 1987), not exact determinants of action. The coordination mechanism is purposely underspecified in these respects. The coordination does not enforce any sequence of problem resolutions that might be implied by the relationships established between bug reports (e.g., a chain of dependent bug reports).
Activity Theory

In activity theory, the unit of analysis is considered to be either an activity system (Engeström, 1999) or an activity (Nardi, 1996b). Nardi defines an activity as being composed of subject, object, actions, and operations (based on her interpretation of Leont’ev, 1974). The subject is an individual or group involved in an activity; the object (as in “the object of the game” (Nardi, 1996b)) is what motivates the activity; actions are object-directed (or, goal-directed) processes enacted to meet the objective. Operations are the constituent parts of actions which can become routine and, with practice, unconsciously performed (e.g., my manipulation of the typewriter keyboard is an unconscious operation in support of the activity of writing this paragraph on activity theory as part of a larger report). Nardi and Engeström both emphasize, in addition, the importance of mediation of activity by artifacts. Nardi writes “artifacts, broadly defined to include instruments, signs, language, and machines, mediate activity and are created by people to control their own behavior” (Nardi, 1996b, p. 75). Artifacts are “persistent structures that stretch across activities through time and space” (p.75). Engeström (1999) assigns to artifacts the role of “break[ing] down the Cartesian walls that isolate the individual mind from the culture and the society” (p.29). This is an important notion in the current project, because the unit of analysis in this project is the bug report, an artifact that we will see is central to SWPM.

Engeström (1999) also emphasizes extensions to the basic triadic model of activity theory that help “explicate the societal and collaborative nature” of peoples’ actions (p.30-31). He uses the term collective activity system to emphasize that the subject is part of a community.

The collective activity system model can be related to elements of the F/OSS community’s SWPM process. The bug report is the artifact; the object is management of software problems and/or bug reports; the subject is the F/OSS community members involved in a particular bug report; the community is the wider group of the users of this community’s software and its developers; the outcome is improved and evolving.
software; division of labor is represented by allocation of responsibility at the artifact (bug report) level to individuals (assignee, reporter, commentator, etc.); the rules are the normative SWPM process and the protocols and conventions enacted by the BRR as a coordination mechanism.

**Distributed Cognition**

The distributed cognition approach to research takes a cognitive system, consisting of people and the artifacts in use, as its unit of analysis. The canonical settings in which this approach has been applied are airplane cockpits (Hutchins & Palen, 1997) and warship navigation (Hutchins, 1995).

According to Hutchins (1995), the division of labor is a fundamental concern of anthropology and a fundamental characteristic of society. Management of software problems is one kind of cognitive work (he distinguishes cognitive work from physical work), and further distinguishes subtypes of cognitive work:

> When the labor that is distributed is cognitive labor, the system involves the distribution of two kinds of cognitive labor: the cognition that is the task and the cognition that governs the coordination of the elements of the task. In such a case, the group performing the cognitive task may have cognitive properties that differ from the cognitive properties of any individual.” (p.176)

Distributed cognition also emphasizes “enactment.” Hutchins stresses that cognition is a cultural process and that culture is not a set of things, but a “process in which our everyday cultural practices are enacted” (Hutchins, 1995) (p.354).

Hutchins positions his work in the context of mainstream cognitive science, so he emphasizes the distinctions between distributed cognition and mainstream cognitive science. He is particularly focused on working against the common “inside/outside” boundary often drawn in cognitive science (e.g., p.355). He also wants to make sure we understand the difference “between the cognitive properties of the sociocultural system
and the cognitive properties of a person who is manipulating the elements of that system.” (Hutchins, 1995) (p.362)

Hutchins (1995) presents a diagram (p. 372) that shows a moment in human practice, where the development of the practice proceeds simultaneously with the development of the practitioners and the conduct of the activity itself. Thus, a software problem is getting fixed while the people working on it are developing as software problem fixes and while the practices of software problem fixing generally are developing.

Distributed cognition has been applied to the study of computer systems in a few areas including software problem repair (Spinuzzi, 2001), computer network troubleshooting (Rogers, 1992), software maintenance (Flor & Hutchins, 1991), and human-computer interaction (Hollan, Hutchins & Kirsh, 2000). So far, however, distributed cognition seems to be most frequently applied to situations characterized by synchronous and collocated activity, like ship navigation and airplane / cockpit operations. Application of distributed cognition to F/OSS SWPM, an asynchronous and distributed activity, may be rare or unique.

Spinuzzi (2001) applies a distributed cognition approach in a paper that compares three analytic frameworks to the same set of data from an observational investigation of software programmers’ use of a large, shared code library during debugging. The three frameworks are contextual design (Beyer & Holtzblatt, 1998), distributed cognition (Hutchins, 1995) and genre ecologies (Spinuzzi & Zachry, 2000). His purpose is to emphasize the strengths and weaknesses of the three analytical approaches to illustrate the contexts in which each might be useful. His application of distributed cognition to a debugging incident he observed while studying software development at Schlumberger illustrates the strengths the distributed cognition approach could have for the current project. He notes that distributed cognition focuses on representations – which can occur in artifacts as well as in the minds of participants within a community – and their computed transformations. The work of fixing software problems depends critically on the development and transformation of representations of software problems: the
software problems themselves are (typically) manifest as behavior, which is not a tangible entity in the common-sense way: a software problem is not an object that you can touch or hold. The people fixing a software problem cannot hold the problematic behavior in their hands: they usually must take certain actions in order to observer the problematic behavior and then work to identify the problematic code; develop hypotheses regarding causes; position the resolution of this particular bug relative to the other demands competing for the organization’s attention (other software problems; requests for new development).

**Organization Type and Information Practices**

Weick and Roberts (1993) develop the idea of the collective mind, which they define as “a pattern of heedful interrelations of actions in a social system” (p.357). They emphasize that collective mind is grounded in the social, that heedful; careful action “constructs mental processes” (p.374) rather than the more common view that mental processes construct actions within groups. The distinctions between high reliability and high efficiency organizations identified by Weick and Roberts suggest an important variable in organizational context. High reliability settings, like the flight deck of an aircraft carrier, demand carefully constructed protocols, rehearsal and slack resources whose availability is used to compensate for unusual contingencies. High efficiency organizations, on the other hand, are organized differently, often in a way that reduces or eliminates slack resources and reduces the need for constant “heedful interrelations.”

Sandusky (2003) showed that different types of work performed in a single setting, a data communications network management organization, were related to different modes and forms of articulation work. When problems occurred in the real-time supervisory control setting, patterns of activity and social order were different from those observed during periods when design work was being done. There was also a relationship between the information practices typical of those engaged in the two different kinds of work in the network management organization. The two predominant types of work
performed in this setting were design work and real-time supervisory control work. Type of work may be another useful organizational variable and an influence on social order, activity and information use.

It remains an open question whether the notion of collective mind, like distributed cognition, can be applied to non-real-time, distributed activities like F/OSS problem management. More research is warranted to examine the differences in a particular kind of work when that work is performed in different types of organizations. One interesting situation would be to compare SWPM in a traditional software development organization to SWPM in a F/OSS development organization. Insights regarding organization type and information, activity and social order influence the part of this project that deals with contexts (discussed in detail in Chapter Six).

Sensemaking

Sensemaking has been used for more than two decades by many researchers studying organizations. Weick presents several conceptions of sensemaking from a variety of sources:

The concept of sensemaking is well named because, literally, it means the making of sense. Active agents construct sensible, sensible (Huber & Daft, 1987, p.154) events. They “structure the unknown” (Waterman, 1990, p.41). How they construct what they construct, why, and with what effects are the central questions for people interested in sensemaking. Investigators who study sensemaking define it in quite different ways. Many investigators (e.g., Dunbar, 1981; Goleman, 1985, pp.197-217) imply what Starbuck and Milliken (1988) make explicit, namely, that sensemaking involves placing stimuli into some kind of framework (p.51). The well-known phrase “frame of reference” has traditionally meant a generalized point of view that directs interpretations (Cantril, 1941, p.20). When people put stimuli into frameworks, this enables them “to comprehend, understand, explain, attribute, extrapolate, and predict” (Starbuck & Milliken, 1988, p.51). (Weick, 1995) (p.4)

Weick also states that “to understand sensemaking is also to understand how people cope with interruptions” (Weick, 1995) (p.5), where Weick implies that interruptions may include situations where “predictions break down.” According to
Weick, some investigators also treat sensemaking as a collective activity engaged in by multiple members of an organization to apply shared understandings, ways of perceiving and acting (Weick, 1995) (p.5).

Weick distinguishes sensemaking from interpretation because interpretation usually focuses on the “reading” of an existing text while sensemaking addresses “how the text is constructed as well as how it is read. Sensemaking is about authoring as well as reading” (p.7). In another section he writes “sensemaking is about the ways people generate what they interpret” and is “less about discovery than it is about inventions” (p.13) (Weick, 1995). Sensemaking emphasizes “an activity or process, where interpretation can be a process but is just as likely to describe a product (p.13).” He also cites Schön’s writings on professional work, especially how professionals encounter problematic situations: they first

convert a problematic situation into a problem…. When [the professional] set[s] the problem, [the professional] select[s] what [he/she] will treat as the ‘things’ of the situation… set the boundaries of … attention to it, and … impose upon it a coherence which allows us to say what is wrong and in what directions the situation needs to be changed. Problem setting is a process in which, interactively, [the professional] name[s] the things to which [he/she] will attend and frame the context in which we will attend to them (Schön, 1983) (p.40), cited in Weick, 1995) (p.9)).

There is implied here a frame, or a structure, which the organization uses to support the work the organization does on the problem. This “frame of reference” helps the organization solve the problem by seeking to make problems similar to each other, if not in all details, at least in terms of their overall framing, components, or structure.

Given this context, it seems that applying sensemaking to the study of software problem management is appropriate. Imagine the submission of a new bug report into a bug report repository where the cause and implications of the problem are not yet clear:

[S]ensemaking seems to address incipient puzzles at an earlier, more tentative stage than does interpretation…. [S]ensemaking begins with the basic question, is it still possible to take things for granted? And if the answer is no, if it has become
impossible to continue with automatic information processing, then the question becomes, why is this so? And, what next? Several questions arise and have to be dealt with before interpretation even comes into play (Weick, 1995) (p.14).

Given the existence of a newly identified bug, the community can no longer take for granted the correct behavior of some part of the software system. The community begins to ask questions: why does this bug manifest? Is this manifestation really a problem? What does it mean to correct this problem (e.g., is it changing one line of code or is the design of the software called into question)? The pre-existing concept of a stable software has broken down.

Weick also emphasizes that sensemaking accounts for the reflexive and often retrospective manner in which people make sense of situations. He relates this back to two sources, Garfinkel (1967) and cognitive dissonance theory’s notion that people retrospectively emphasize positive precursor events or situations when outcomes are positive and emphasize negative precursors when outcomes are negative.

The use of Weick’s sensemaking is, however, problematic in some regards. This section examines Weick’s seven properties of sensemaking (Weick, 1995, chapter 2) one by one in order to come to illustrate the appropriateness of applying sensemaking to the analysis of software problem management.

1. **Sensemaking is grounded in identity construction.** Weick emphasizes that it is wrong to assume only a single identity is under construction during the sensemaking process. Nor is identity associated only with an individual: organizations also have identities that are constructed. Perhaps the best way to think about identity construction in the context of software problem management is to consider how the software development organization’s own identity is affected by its own software problem management process. An effective, healthy process should result in improved software and thus, indirectly, in an improved organizational identity. The organization itself is viewed as an organization that enacts (embodies) more or less quality. I’d also argue that participation by
community members affects their own identities, too. For example, how the quality of one’s own work affects one’s reputation.

2. *Sensemaking is retrospective.* We can only make sense of a situation by examining and reflecting upon what has already occurred. The identified software problem is in the system now and has been for some time, maybe for a long time. We can only look back at the code that already exists, the situations of use that have occurred and caused the bug to manifest to help us correct the problem. By looking back, we can construct the hypotheses about problem cause and by looking back we can plan steps to confirm or rule out parts or all of our explanatory cause. “Whatever is now, at the present moment, under way will determine the meaning of whatever has … occurred” (Weick, 1995) (p.27). In sensemaking … “many possible meanings may need to be synthesized….there are too many meanings, not too few. The problem faced by the sensemaker is one of equivocality, not one of uncertainty. The problem is confusion, not ignorance” (Weick, 1995) (p.27).

3. *Sensemaking is enactive of sensible environments.* Enactment means “in organizational life, people often produce part of the environment they face” (Weick, p. 30). “[T]here is not some kind of monolithic, singular, fixed environment that exists detached from and external to” the people within some organization (p.31). “They act, and in doing so create the materials that become the constraints and opportunities they face.” (p.31) Weick cautions to be aware of the false and misleading dichotomy of cause then effect or stimulus then response. Instead, we should remember enactment and that sensemaking is an activity of relating things (events or objects) to each other.
4. *Sensemaking is social.* Weick quotes Walsh and Ungson (1991): “An organization is ‘a network of intersubjectively shared meanings that are sustained through the development and use of a common language and everyday social interactions.’” (p. 60 in Walsh & Ungson, p. 39 in Weick) Sensemaking is an activity, and, within organizations, sensemaking activities affect and are contingent upon the activities of others. Weick quotes March and Olsen (1976) when they describe organizations as “a set of procedures for argumentation and interpretation” (p.25 in March & Olsen, p. 41 in Weick).

5. *Sensemaking is ongoing.* It never really has a starting point and an end point. People are always in the middle of complex situations that have to be dealt with. Weick discusses how people and organizations often are involved in performing “organized action sequences.” Opportunities for sensemaking arise when one of these sequences is interrupted (imagine an unexpected result in the middle of execution of a Standard Operating Procedure, or the identification of a bug during a software release).

6. *Sensemaking is focused on and by extracted cues.* To take a sensemaking approach, you need to look at how people deal with problems and unexpected events. Look at how they “notice, extract cues, and embellish that which they extract…. Control over which cues will serve as a point of reference is an important source of power.” (p.50) “A specific observation becomes linked with a more general form or idea in the interest of sensemaking, which then clarifies the meaning of the particular, which then alters slightly the general, and so on. The abstract and the concrete inform and construct one another.” (p.51) “Context affects what is extracted as a cue in the first place…. Context also affects how the extracted cue is then interpreted.” (p.51) Indexicality plays a role
in how cues are handled, too, like indexicals in spoken language. The idea that any plan (not necessarily the best, optimal or most accurate) may do: “… plans are a lot like maps. They animate and orient people. Once people begin to act (enactment), they generate tangible outcomes (cues) in some context (social), and this helps them discover (retrospect) what is occurring (ongoing), what needs to be explained (plausibility), and what should be done next (identity enhancement).” (p.55) So once action starts, the person /organization enacts and influences what else happens.

7. Sensemaking is driven by plausibility rather than accuracy. Many studies of behavior in organizations show that accuracy, e.g., in terms of accuracy of information, is not required in order for people to act. Weick cites some of the following as reasons why. (1) People need to filter and edit in order not to be overwhelmed by data. (2) Typically, people need to (and should) begin to act before perfect information is available. “It is more crucial to get some interpretation to start with than to postpone action until ‘the’ interpretation surfaces” (p.57). (3) Quick responses allow actors (people or organizations) to enact or influence events as they unfold. “A fast response can be an influential response that enacts an environment.” (p.58) (4) Accuracy, when important, is usually only an issue for limited time and over limited pieces of data. Different kinds of information may need to be more accurate than other information depending upon the situations. (5) Sensemaking is usually concerned with the interpersonal, interactive, interdependent qualities of organizational life rather than the perception of fixed objects. Entities like people or other organizations are never fixed: they mutate as situations unfold, making it impossible to create accurate mental representations of such entities. (6) The frame, or area, within which people act or try to make sense is bounded (e.g., a project or a certain bug report). Accuracy only matters - and only matters
sometimes - within this limited frame. (7) People interested in acting are usually more interested in filtering and simplifying stimuli rather than have more stimuli, where more stimuli implies a more accurate representation of some thing or situation. (8) You cannot tell when you perceive something if that perception will prove to be accurate over the long run. “What is necessary in sensemaking is a good story.” (p.61) “A good story holds disparate elements together long enough to energize and guide action, plausibly enough to allow people to make retrospective sense of whatever happens, and engagingly enough that others will contribute their own inputs in the interest of sensemaking.” (p.61)

**Coordination/Cooperation Theories**

Schmidt and Simone (1996) define a coordination mechanism as “a construct consisting of a *coordinative protocol* (an integrated set of procedures and conventions stipulating the articulation of interdependent distributed activities) on the one hand and on the other hand *an artifact* (a permanent symbolic construct) in which the protocol is objectified) [emphasis in original] (p.165-166).

Crowston (1997) has developed coordination theory, which “is intended to analyze organizations in a way that facilitates redesign” of organizational processes (p.161). Coordination theory has as its focus *activities and goals (tasks)*, *actors*, and *resources*. One reason to apply coordination theory to organizational analysis is to ascertain why a process is structured the way it is and to identify alternatives to the existing process. As Schmidt and Simone (1996) and other researchers do, Crowston makes a distinction between the activities “that are necessary to achieve the goal of the process (e.g., that directly contribute to the output of the process) and those that serve … to manage various dependencies between activities and resources” (pp. 159-160). Schmidt and Simone refer to the two types of work as the “common field of work” and “articulation
work” (p.157); Crowston uses the term “coordination mechanism” instead of “articulation work” (p.160).

Berg & Bowker (1997) describe the coordinating properties of medical records, arguing that they play multiple roles in clinical settings. While the medical record itself represents the patient in the hospital, it also constructs the patient because it is a central source of information for decisions on how the patient’s treatment will proceed. The record also plays a role in representing the work of multiple constituencies in the clinic (doctors, nurses and labs) and the organizational structure of the clinic itself. Finally, the medical record acts as a composite, or information container, that holds many different kinds of discrete records: vital signs; doctors’ notes; x-rays; lab reports.

Medical records are an important object of research because they “…occup[y] a central niche in this network: it is where many of the nurses’ and physician’s tasks begin, end, and are coordinated, where inscriptions accumulate…” (p.514). Berg and Bowker examine the entities (“bodies politic”) produced by medical records: the clinic; the doctors; the nurses. Their analysis suggests a correspondence between the SWPM work and medical work. The bug reports are similar to medical records; the software problems have some correspondence to the bodies acted upon in the clinic; this F/OSS community corresponds in some sense to the clinic; the members of this F/OSS community, forming parties with particular interests in the bug reports like quality assurance workers, coders, and managers, roughly correspond to doctors, nurses, hospital management, and insurance companies; bug trajectory corresponds to illness trajectory.

Berg & Bowker show that the medical record describes “…past action in the context of a set of organizational arrangements” (p.523). The purpose of this study is to bring into focus what bug reports can tell us about the work of assuring software quality – the past action, or how software problems are actually managed; the ways in which that work is organized or aligned and who participates – the set of organizational arrangements. Although it is beyond the scope of this project, it is interesting to consider how diagnostic records and the information practices associated with them may be similar in a variety of domains – medicine, various branches of engineering (e.g., the
manufacture of automobiles, aerospace devices, bridges and buildings, etc.) as well as software engineering.

An Approach to the Study of Information Practices in Distributed Work

The final section of this chapter summarizes the earlier sections on research in the domain and social informatics compatible approaches to the study of information, activity and social order in a large distributed work community and what that research reveals to us in terms of conclusions, methods and theory. Limitations of previous research and the remaining open questions are also discussed. Finally, the approach selected for this project is described and justified, with an emphasis on the unique synthesis of human information behavior approaches with methods typically employed in social informatics research.

The review of the literature (above) has highlighted some of the previous research in the domain of software quality and software problem management, including software development, the relationships between organizational structure and software process and quality, emerging F/OSS development processes, software problem repair at an individual level of analysis, and software problem management as a socio-technical process. Research from a software engineering perspective has been dominated by the quest for generalized, effective quantitative models that would allow software development organizations to detect software problems as early in the development process as possible, support the efficient categorization and prioritization of the problems and the repair of those problems (Osterweil 1996). Other research, comprising the minority of research in the domain, has revealed the critical importance of organizations, individuals and processes on software quality (e.g., Conway, 1968; Krasner, Curtis & Iscoe, 1987; Brooks, 1995; Grinter, 1995; Herbsleb et.al., 1997; Herbsleb & Grinter, 1999). The emergence of F/OSS software development practices has promoted more researchers to take social and organizational factors into account in their research, making claims (as yet unproven) that the nature of the social organization of these projects leads
to the creation of higher quality software (e.g., Raymond, 1999). Finally, a few researchers have looked at software problem management as a socio-technical process, but have not fully described the varieties of information at play, the processes employed, the embedding social order, and the contexts bearing upon the information, processes and social order (Carstensen, Sørensen & Tuikka, 1995; Crowston, 1997; Kajko-Mattsson, 1999). These particular projects have examined software problem management not in an effort to fully understand it but to use software problem management as a means to address some other issue, such as the development of a general model to provide computer-based support for articulation work (Carstensen, Sørensen & Tuikka, 1995); to develop a general theory of coordination (Crowston, 1997); or to simply provide a detailed description of current practices of software problem management and software maintenance in two organizations (Kajko-Mattsson, 1999).

Several potential methods for studying software problem management from a social informatics perspective were described earlier in this chapter as well. Many of these approaches are compatible with a social informatics approach and have been employed previously in research associated with fields such as computer-supported cooperative work (CSCW), participatory design and studies of organizations and their use of information. For example, Egger & Wagner (1992) applied Strauss’ theory of negotiation to the design of information systems intended to support time management and scheduling of surgical teams. Strauss’ conceptualization of articulation work has been extraordinarily influential in computer-supported cooperated work (e.g., Schmidt & Simone, 1996). Activity theory has often been applied to research in software and systems design (Nardi, 1996a). The notions of distributed cognition (Hutchins, 1995) and collective mind (Weick & Roberts, 1993) have been used by researchers in CSCW to explain the observations of how groups work together, particularly in real-time, co-located environments, like air traffic control centers (Randall, Hughes & Shapiro, 1994).

The earlier sections emphasized a body of research that is now commonly referred to by the term human information behavior. This research tradition, based in Library and Information Science (LIS), has already provided important conclusions, models and
theory regarding how individuals seek, search for, acquire and use information. This body of work has begun to expand its reach from its earlier focus on traditional library-based information systems out toward a range of information behaviors reflecting the variety of activities and needs of human beings in work settings, social settings and their communities (Savolainen, 1995; Bishop et. al, 2000; Bishop et. al., 2003; McKenzie, 2003; McKenzie, 2004; Fisher, Durrance & Hinton, 2004).

No comprehensive description of software problem management as a socio-technical system in any domain has yet been created. This project uses a social informatics approach to build the first description of this type based upon the practices of one F/OSS development community through the careful examination of the bug reports created to support the management of software problems. This approach has the benefits of revealing subtle aspects of the information used by the community as they manage the constant daily flow of hundreds of bug reports and the problems, design issues and red herrings (that is, the non-problems) they represent. This approach also reveals important details about the layered and inter-related contexts that influence human information behavior, social order and processes; it exposes the dynamic and contingent nature of the social order that supports this critical community activity. By carefully revealing these details, we can begin building testable hypotheses and models in order to support the development of a general theory of software problem management as a complex socio-technical assemblage, which has never been attempted before.

Human information behavior research is rapidly expanding its reach. In the past, it has often been limited in its scope. Studies in LIS often limit their object of study to the retrieval of specific forms of information from specialized digital libraries, abstracting and indexing services and on-line public access catalogs (OPACs). Often the range of human information behaviors considered is constrained to information searching or more generally information seeking. Recently, information use has become a common object of human information behavior research, an important and exciting development in the field. Another limitation is that much of the previous information behavior research has been done on “types” of users: engineers, scientists, doctors, pregnant women. An
underlying assumption is that any individual fitting the type will exhibit information behaviors that share significant commonalities. “This framework assumes that by studying the groups to which the users belong, researchers can determine the needs of the individual members of the group” (p.148) (Hewins, 1990). However, while the range of types of users studied is widening in recent years, the focus is still predominantly on the activities of isolated individuals instead of the information behaviors and practices of groups of people.

This project draws primarily from the following research approaches described above:

- Human information behavior
- Theories of negotiation, context, and negotiated order
- Information practices
- Sensemaking
- Coordination/Cooperation Theories

These approaches, described in the section Approaches to the Study of Information, Activity and Social Order in Distributed Work, are used to help interpret the data, create the map of SWPM work and interpret that map. Concepts from human information behavior are used in identifying the information objects present, used, not used and created as the community manages software problems. Sensemaking, in Weick’s sense of the term, is used as an over-arching conceptualization of the process of managing software problems. Several theoretical constructs associated with the work of Anselm Strauss and his colleagues are employed throughout the analysis to help describe and explain the phenomena found through the examination of this community’s bug reports. Principal theories from Strauss applied in this project include negotiation, negotiated order, contexts, and to a lesser extent, trajectories. Theories of coordination and cooperation are used in particular to help explain the mutually constitutive roles of
technology and process as well as to help understand the how this community dynamically organizes actors in this distributed environment.

Theories of cognition, particularly distributed cognition and collective mind, while having intriguing possibilities for interpretation of the data discussed in subsequent chapters, is generally not applied here. Virtually all of the work done with distributed cognition has been done in settings typified by co-location of participants in real-time settings (aircraft carrier flight decks, ship navigation, airplane cockpits, etc.). Adapting and applying these theories to situations where participants are not co-located and their activities are generally asynchronous is left for the future and for other researchers.

Activity theory, likewise, is not applied in the subsequent analysis. Activity theory could be applied in this analysis, but was left for future application in the interests of making the subsequent chapters understandable and tractable. One of the main goals of this project is the creation of a map of SWPM, necessitating an engagement with the data that is at a moderate level of granularity. Activity theory would likely be quite useful in detailed, fine-grained analyses of the community’s interactions around particular bug reports.

The LIS human information behavior research is beginning to incorporate wider conceptions of users, information behaviors, situations of use, contexts and types of information than was typical of the period preceding Dervin & Nilan’s seminal paper (1986). Examining information use and the construction of information is just beginning to be incorporated in studies of this type (Wilson, 1999; Bishop et. al., 2003; McKenzie, 2003; McKenzie, 2004; Fisher, Durrance & Hinton, 2004). Bishop et. al. and McKenzie, in particular, acknowledge the centrality of collective or community behavior in adopting a holistic view of information behavior.

Wilson (1999) proposes a nested model of human information behavior research within which information-seeking behavior is one type of information behavior, and within which information search behavior is one type of information-seeking behavior. This model, however, is focused on the information behavior of individuals. McKenzie (2003) proposes using the term information practices to emphasize the social nature of
information behavior in co-located group or community settings. Thus information practices comprise the information behavior of multiple people. Another layer can be added to this model by considering the more complex issues of distributed (collective) information practices, which are the object of the current project (see Figure 2-1).

Figure 2-1 Model of Human Information Behavior / Information Practices
(based upon Wilson, 1999)

The emerging theory associated with human information behavior research can benefit from integration with research approaches commonly used in the broader field of social informatics. This is necessary in order to both generalize information behavior theories and to extend them to accommodate the practices of distributed work communities. The traditions of research from CSCW and participatory design are useful
because they provide insights on how groups of people with heterogeneous backgrounds perform information intensive work. Approaches common in CSCW and participatory design have also been applied, if rarely, to the examination of the complex domain of software development. The recent emergence of F/OSS development practices has revolutionized some aspects of software development, particularly in the extreme degree of geographic distribution of participants. (See Herbsleb & Grinter, 1999 for an excellent description of globally distributed software development in a traditional, commercial software development context. This level of geographic distribution of work is also important, but is an evolutionary step from traditional co-located software development practice.)

This project builds on these established traditions in LIS, CSCW, etc., in order to provide basic, empirically grounded insights into SWPM, a kind of work of critical importance to modern society; to assist in the development of improved tools and techniques for SWPM work; and to provide advances in our overall understanding of human information behavior and information practices in distributed work settings. In contrast to the focus of many information behavior studies in LIS on homogeneous groups of people, like aerospace engineers, this project examines a large, heterogeneous, geographically distributed group of people who are non-specialist users of the software; volunteer software developers; software developers who are part of the development effort because their employers deem that to be their role in their organization; leaders in the development community who guide the overall design and evolution of the system; quality assurance specialists who are likely to be less technically skilled than the developers and community leaders, human-computer interface designers, etc.

This project attempts to take human information behavior and social informatics research to a new level, to examine the collective information practices of this F/OSS development community. This is an appropriate research approach because we need a better understanding human information behavior at an individual level, information practices at a collective level and distributed collective information practices at the most general level (Sandusky, Gasser & Ripoche, 2004b). What has been learned as part of this
project can contribute to a general theory of human information behavior at the level of individuals and at the level of collective information practices in both co-located and distributed settings.
CHAPTER THREE: METHODS

Introduction

This chapter describes the research setting and the methods employed to gather and analyze bug reports for this project. The chapter begins with a description of the research setting. The research approach – employing document / content analysis and grounded theory – is described next. Then the sampling process is discussed and the fit between the population and sample is demonstrated. Next, I describe the data collection and preparation process. The coding and analysis processes are described in the following section. Finally, the validity, reliability, strengths and limitations of the methods employed here are discussed.

Research Setting

This project was conducted in the context of a large-scale distributed F/OSS development project organized responsible for producing a set of twenty-four software products and sub-products.

This community’s mission statement lists the following responsibilities focused on centralized code integration:

- Provide technical and architectural direction for the project
- Collect and synchronize code changes
- Create source code releases
- Operate discussion forums
- Coordinate bug management (that is, perform software problem management)
- Provide documentation (code “road maps”)
- Help people achieve consensus
Organization

Like many F/OSS projects, large and small, the project whose SWPM process is examined here depends upon a large, distributed group of people to perform most of the work: design, coding, SWPM while a small, core group manages the project web site, runs the bug report repository and code repositories, etc. This model is sometimes termed “distributed development with centralized integration.”

Major code sub-systems (e.g., each different unit termed a “product” in this community) has a “module owner” who is responsible for making the ultimate decision on what code goes into and what code stays out of a particular release of a particular sub-system or product. At the time this was written (in early 2004), there were 90 distinct “module owner” positions. In some cases, no one was actually defined as being the module owner. Some modules have module “co-owners.” Some individuals own more than one module.

Research Approach

The research approach taken in this project combines document / content analysis with grounded theory to support the identification of and categorization of activities, forms of information, processes, social forms, and artifacts related to the work of resolving software problems, as those concepts and categories are warranted by the bug reports analyzed. The entire project is guided by the tenets of social informatics: individuals, information, technology, organization, and context together form an inseparable ensemble. Theories of negotiation and negotiated order (Strauss, 1978); theories of coordination (Schmidt & Simone, 1996; Crowston, 1997); distributed cognition (Hutchins, 1995), and sensemaking (Weick, 1995) are also applied to develop an understanding of information, activity, and social order revealed by the analysis of the bug reports.
Document / Content Analysis

Document analysis can be performed on “… written materials that contain information about the phenomena we wish to study” (Bailey, 1994) (p.294). Document analysis can be performed on either primary or secondary documents. In this study, the bug reports are primary documents generated by this F/OSS development community to help organize, track and control the resolution of software problems, or bugs.

Document analysis has a number of advantages and disadvantages (see Bailey, 1994, pp. 294-298 for a thorough discussion). In the context of this study, the bug reports provide certain characteristics that support this research. (1) The bug reports contain information characterizing the work done to correct software problems over a range of several years, allowing examination of bug repair activity that occurred years ago; (2) study of bug reports avoids reactivity, or a change in behavior due to the presence of a researcher (also known as the Hawthorne effect), on the part of the subjects under study; (3) the bug reports capture the thoughts of the community members at the time their thoughts, opinions and findings were fresh, and are not based upon or filtered by memory; (4) a consistent format and consistent social order within which the bug reports are created and used, enabling me to compare information in bug reports created several years apart.

Conversely, the bug reports have some problematic characteristics common to many document-based studies. (1) They do not necessarily record all the information used, interactions regarding, or media used in the resolution of a particular software problem: important, often contextual or social interactions, like email, face-to-face, telephone or IRC are not represented in the bug reports; (2) the bug reports contain a lot of unstructured text, which makes consistent coding of the text difficult; (3) differences between the content of bug reports created or amended at different points in time may not be apparent in the documents themselves but may have been caused by changes in
organizational processes or technology (e.g., a change in the structure of the bug report could mean that an important piece of data is included in only a subset of all bug reports).

Techniques from content analysis are also applied. Content analysis is a method used to “…make valid inferences from text” (Weber, 1990) (p.9). In this project, the texts are the bug reports taken from the bug report repository maintained by a large, thriving F/OSS development community. Content analysis is used to organize the ways information is expressed in bug reports into a smaller number of categories (Weber, 1990) (p.12). Content analysis is often used to support the conversion of qualitative data into quantitative data in order to support hypothesis testing (Bailey, 1994) (p.304). Typically a researcher “…first constructs a set of mutually exclusive and exhaustive categories that can be used to analyze documents, and then records the frequency with which each of these categories is observed in the documents studied” (Bailey, 1994, p.305). While I may report frequencies of the presence of particular categories in the bug report data, my goal is not to test hypotheses or produce extensive quantitative data summaries. Instead, I am performing a predominately, but not exclusively, qualitative research project. Many forms of content analysis, especially those that go beyond simple word counting exercises and include application of human intellectual effort to the development of categories, include an important qualitative component. Categories relevant to the research project must be drawn from the content of the documents in order to support analysis and development of theory. These categories will be, ideally, “…exhaustive, mutually exclusive and independent. By ‘independent’ we mean that the value of one category does not determine the value of another category” (Bailey, 1994) (p.306).

**Grounded Theory**

As a method, content analysis allows room for application of many procedures for the development of relevant categories. I used grounded theory (Strauss & Corbin, 1990) procedures (specifically the procedure known as open coding – see Strauss &
Corbin, Chapter 5) to inductively derive the categories that emerge from the data contained in the bug reports. Each bug report was examined in order to identify various conceptual labels warranted by the data. As the analysis proceeded, the conceptual labels were combined into a smaller number of categories. The properties and dimensions of each of the categories were also identified during open coding.

Open coding is the first analytic step in grounded theory. It is a process whereby you “…nam[e] and categoriz[e] phenomena through close examination of data” (Strauss & Corbin, 1990) (p.62). The process involves iterations of examining the data, using any of a variety of techniques (e.g., document analysis; observation), noting concepts that appear in the data, and analysis of the concepts. The result of this process should be a provisional set of concepts, organized into categories, with each category including properties with dimensions.

For example, Strauss and Corbin use observing work in a restaurant as an example of how open coding works. They identify concepts such as work site, information passing, conferring and watching. Then concepts are grouped into categories. One category is types of work, to which the concept watching belongs. Relevant dimensions for each category and concept can be defined along with dimensions to allow comparison. Watching, for example, may have properties like frequency, with dimensional points like “never” and “often,” and intensity, with dimensional points like ‘high’ and ‘low’ (Strauss & Corbin, 1990, Chapter 5). The names assigned to concepts and categories may be borrowed from other research or theories, but many are likely to be suggested by the language used by members of the community being researched.

In this project, I develop concepts and categories from data primarily found in the community’s bug reports, held in the community’s bug report repository. Categories developed through the examination of the bug reports include information objects, activities, processes, social order, and contexts. Concepts belonging to the category information objects include bug reports, bug report networks, dialogs, and statements. Concepts belonging to the category activities include reviewing, verifying, confirming, describing, requesting, and verifying. The category processes includes concepts like negotiation, triage, sensemaking, and voting. Social
order has associated concepts like *dynamic distributed cooperative ensemble* and *roles*. Concepts belonging to contexts include *structural context, technical context, economic context*, and *bug report context*.

Axial coding is the next major analytic step in grounded theory. During axial coding, “…data are put together in new ways after open coding, by making connections between categories. This is done by utilizing a coding paradigm involving conditions, context, action/interactional strategies and consequences” (Strauss & Corbin, 1990) (p. 96). In this project, the connections between categories and specific concepts within categories are described in later chapters, particularly Chapter Nine.

Grounded theory is an appropriate method to apply in this project because the method’s objective is to allow the researcher to develop insights, and ultimately build theory, out of the raw material drawn from qualitative data. In the context of this project, there was no existing theory of software problems, representations of software problems or software problem work within open source software communities. Therefore, I didn’t know what the relevant concepts, categories, properties and dimensions were for this project. In exploring a type of work and its artifacts that has never been studied in this way, the concepts and categories must be systematically drawn from the raw data, in this project, the bug reports created and managed by this F/OSS development community. Grounded theory was expressly designed to support this type of inquiry.

**Sampling**

The Population

The population – the total of all cases that conform to the necessary criteria – of bug reports consists of those bug reports held in this F/OSS community’s bug report repository with “Opened” dates between 04/07/1998 and 12/01/2002). There are more than 180,000 bug reports with “opened” dates falling in the date range. The bug reports in the repository are representations of “bugs,” here called software problems, identified
with any of the products supported by this F/OSS development community. As of 03/01/2004, there were more than 235,000 bug reports in the community’s bug report repository. Bug reports are added to the repository at a rapid rate (about 140 per day): 4,458 in December 2002, 4,221 in January 2003, 3,974 in February 2003 and 4,564 in March 2003. When each bug report is created, it is assigned a serial number, the bug report number.

There are an undetermined number of bug reports that are missing from the repository for unknown reasons. There are many missing bug reports from the lower end of the range of bug report numbers, when this bug report repository was presumably a new system. For example, bug reports 1-34 do not exist, and are referred to as missing bug reports in this project. These records may have been created as test records, or were problematic and therefore physically deleted from the repository. There are other bug reports in the repository that cannot be retrieved from the public repository interface because of access restrictions; an account with appropriate privileges is required to access such access-restricted bug reports. The total number of missing and access-restricted bug reports in the population is not known.

The Table 3-1 (below) shows the rate at which bug reports have been added to the community’s bug report repository per month since April 1998. Selected products, those most frequently cited in bug reports, are broken out in separate columns.

<table>
<thead>
<tr>
<th>Year, Month</th>
<th>Product A</th>
<th>Product B</th>
<th>Product D</th>
<th>Product C</th>
<th>Total Bug Reports Opened</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998-04</td>
<td>27</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>180</td>
</tr>
<tr>
<td>1998-05</td>
<td>24</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>105</td>
</tr>
<tr>
<td>1998-06</td>
<td>19</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>61</td>
</tr>
<tr>
<td>1998-07</td>
<td>34</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>53</td>
</tr>
<tr>
<td>1998-08</td>
<td>132</td>
<td>0</td>
<td>1</td>
<td>13</td>
<td>166</td>
</tr>
<tr>
<td>1998-09</td>
<td>141</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>256</td>
</tr>
</tbody>
</table>

6 Bug reports can be marked if they are related to security issues (users’ private information revealed; users’ systems become vulnerable to data destruction or be used to mount attacks on other systems). These bug reports cannot be retrieved from the bug report repository unless the repository user has the appropriate level of authorization.
### The Bug Report Population: Distribution by Most Popular Product Values

<table>
<thead>
<tr>
<th>Year, Month</th>
<th>Product A</th>
<th>Product B</th>
<th>Product D</th>
<th>Product C</th>
<th>Total Bug Reports Opened</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998-10</td>
<td>236</td>
<td>13</td>
<td>0</td>
<td>5</td>
<td>292</td>
</tr>
<tr>
<td>1998-11</td>
<td>400</td>
<td>13</td>
<td>0</td>
<td>5</td>
<td>489</td>
</tr>
<tr>
<td>1998-12</td>
<td>360</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>407</td>
</tr>
<tr>
<td>1999-01</td>
<td>595</td>
<td>23</td>
<td>2</td>
<td>31</td>
<td>689</td>
</tr>
<tr>
<td>1999-02</td>
<td>406</td>
<td>10</td>
<td>2</td>
<td>29</td>
<td>484</td>
</tr>
<tr>
<td>1999-03</td>
<td>932</td>
<td>53</td>
<td>5</td>
<td>31</td>
<td>1106</td>
</tr>
<tr>
<td>1999-04</td>
<td>1069</td>
<td>126</td>
<td>5</td>
<td>31</td>
<td>1314</td>
</tr>
<tr>
<td>1999-05</td>
<td>1301</td>
<td>140</td>
<td>1</td>
<td>42</td>
<td>1544</td>
</tr>
<tr>
<td>1999-06</td>
<td>1388</td>
<td>220</td>
<td>10</td>
<td>41</td>
<td>1708</td>
</tr>
<tr>
<td>1999-07</td>
<td>1542</td>
<td>294</td>
<td>1</td>
<td>21</td>
<td>1893</td>
</tr>
<tr>
<td>1999-08</td>
<td>1511</td>
<td>301</td>
<td>0</td>
<td>26</td>
<td>1884</td>
</tr>
<tr>
<td>1999-09</td>
<td>1828</td>
<td>414</td>
<td>2</td>
<td>28</td>
<td>2352</td>
</tr>
<tr>
<td>1999-10</td>
<td>1790</td>
<td>347</td>
<td>4</td>
<td>48</td>
<td>2287</td>
</tr>
<tr>
<td>1999-11</td>
<td>2190</td>
<td>432</td>
<td>4</td>
<td>33</td>
<td>2754</td>
</tr>
<tr>
<td>1999-12</td>
<td>1882</td>
<td>427</td>
<td>11</td>
<td>32</td>
<td>2390</td>
</tr>
<tr>
<td>2000-01</td>
<td>2342</td>
<td>581</td>
<td>15</td>
<td>81</td>
<td>3118</td>
</tr>
<tr>
<td>2000-02</td>
<td>2845</td>
<td>578</td>
<td>6</td>
<td>74</td>
<td>3625</td>
</tr>
<tr>
<td>2000-03</td>
<td>3266</td>
<td>593</td>
<td>15</td>
<td>101</td>
<td>4176</td>
</tr>
<tr>
<td>2000-04</td>
<td>2654</td>
<td>476</td>
<td>18</td>
<td>44</td>
<td>3317</td>
</tr>
<tr>
<td>2000-05</td>
<td>2546</td>
<td>608</td>
<td>5</td>
<td>85</td>
<td>3406</td>
</tr>
<tr>
<td>2000-06</td>
<td>2399</td>
<td>544</td>
<td>8</td>
<td>69</td>
<td>3166</td>
</tr>
<tr>
<td>2000-07</td>
<td>2039</td>
<td>500</td>
<td>9</td>
<td>51</td>
<td>2718</td>
</tr>
<tr>
<td>2000-08</td>
<td>3018</td>
<td>551</td>
<td>16</td>
<td>43</td>
<td>3824</td>
</tr>
<tr>
<td>2000-09</td>
<td>2988</td>
<td>456</td>
<td>25</td>
<td>70</td>
<td>3786</td>
</tr>
<tr>
<td>2000-10</td>
<td>2932</td>
<td>582</td>
<td>69</td>
<td>43</td>
<td>3848</td>
</tr>
<tr>
<td>2000-11</td>
<td>2209</td>
<td>508</td>
<td>74</td>
<td>35</td>
<td>2976</td>
</tr>
<tr>
<td>2000-12</td>
<td>1671</td>
<td>363</td>
<td>43</td>
<td>44</td>
<td>2322</td>
</tr>
<tr>
<td>2001-01</td>
<td>2294</td>
<td>596</td>
<td>68</td>
<td>72</td>
<td>3228</td>
</tr>
<tr>
<td>2001-02</td>
<td>2402</td>
<td>564</td>
<td>49</td>
<td>71</td>
<td>3274</td>
</tr>
<tr>
<td>2001-03</td>
<td>2582</td>
<td>739</td>
<td>60</td>
<td>113</td>
<td>3714</td>
</tr>
<tr>
<td>2001-04</td>
<td>2825</td>
<td>723</td>
<td>45</td>
<td>74</td>
<td>4024</td>
</tr>
<tr>
<td>2001-05</td>
<td>3004</td>
<td>974</td>
<td>82</td>
<td>76</td>
<td>5297</td>
</tr>
<tr>
<td>2001-06</td>
<td>3587</td>
<td>783</td>
<td>191</td>
<td>93</td>
<td>5010</td>
</tr>
<tr>
<td>2001-07</td>
<td>3012</td>
<td>783</td>
<td>199</td>
<td>77</td>
<td>4383</td>
</tr>
<tr>
<td>2001-08</td>
<td>3277</td>
<td>818</td>
<td>234</td>
<td>192</td>
<td>4840</td>
</tr>
<tr>
<td>2001-09</td>
<td>3196</td>
<td>765</td>
<td>191</td>
<td>168</td>
<td>4580</td>
</tr>
<tr>
<td>2001-10</td>
<td>3764</td>
<td>1001</td>
<td>211</td>
<td>173</td>
<td>5444</td>
</tr>
<tr>
<td>2001-11</td>
<td>3555</td>
<td>912</td>
<td>185</td>
<td>147</td>
<td>5077</td>
</tr>
<tr>
<td>2001-12</td>
<td>3289</td>
<td>693</td>
<td>198</td>
<td>69</td>
<td>4567</td>
</tr>
<tr>
<td>2002-01</td>
<td>3660</td>
<td>916</td>
<td>183</td>
<td>142</td>
<td>5342</td>
</tr>
<tr>
<td>2002-02</td>
<td>3746</td>
<td>1008</td>
<td>219</td>
<td>140</td>
<td>5459</td>
</tr>
<tr>
<td>2002-03</td>
<td>4261</td>
<td>1063</td>
<td>286</td>
<td>144</td>
<td>6190</td>
</tr>
<tr>
<td>2002-04</td>
<td>4710</td>
<td>1165</td>
<td>258</td>
<td>205</td>
<td>6803</td>
</tr>
<tr>
<td>2002-05</td>
<td>4944</td>
<td>1171</td>
<td>326</td>
<td>161</td>
<td>7076</td>
</tr>
<tr>
<td>2002-06</td>
<td>4379</td>
<td>913</td>
<td>336</td>
<td>210</td>
<td>6653</td>
</tr>
<tr>
<td>2002-07</td>
<td>3422</td>
<td>760</td>
<td>268</td>
<td>223</td>
<td>5317</td>
</tr>
<tr>
<td>2002-08</td>
<td>3612</td>
<td>723</td>
<td>243</td>
<td>157</td>
<td>5516</td>
</tr>
<tr>
<td>2002-09</td>
<td>3583</td>
<td>862</td>
<td>223</td>
<td>148</td>
<td>5865</td>
</tr>
<tr>
<td>2002-10</td>
<td>3439</td>
<td>891</td>
<td>204</td>
<td>223</td>
<td>5987</td>
</tr>
<tr>
<td>2002-11</td>
<td>3793</td>
<td>793</td>
<td>159</td>
<td>252</td>
<td>4980</td>
</tr>
</tbody>
</table>

**Totals**: 128,842, 28,772, 4,773, 4,524, 181,322
Systematic Random Sampling

Systematic random sampling was used to draw the sample from the population. To perform systematic random sampling, you:

1. determine the appropriate sample size
2. determine a sampling ratio based upon the size of the population and the sample size
3. select an arbitrary or randomly selected starting point within the population
4. select every Nth instance from the population for inclusion in the sample

The sample size was determined using the table developed by Krejcie and Morgan (1970) (reprinted in Powell, 1991, p. 75) for determining appropriate sample sizes from a given population. This may be, as noted by Powell (1991), “… a rather simplistic, and quite possibly conservative, method for ascertaining a sample size.” (p.75) However, it makes it possible to identify a reasonable sample size quickly without performing the statistical analysis necessary to determine the variability of the population. Such analysis is difficult, if not impossible, in the context of this project, whose goal is to identify bug report trajectories based upon qualitative analysis of the bug reports themselves. At this point, we don’t know the variability of all characteristics of bug reports, nor do we know which characteristics are significant with regard to bug trajectories. A conservatively large sample size helps ensure that the sample will include a representative variety of the kinds of bug reports that exist in the community’s bug report repository.

The table of sample sizes (Powell, 1991) includes the following break points. For a population (N) of 75,000, the appropriate sample size is 382. For N = 1,000,000, the appropriate sample size is 384. I chose a sample size of 385 in order to account for the possibility that the sample might include some bugs that do not exist in the repository – missing bug reports – or are not available for use due to access restrictions – access restricted bug reports - as described above.
Systematic sampling has the potential disadvantage of bias introduced by patterns within the population. The population here is essentially a list of bug reports, ordered by the date and time they each were opened. We assume that the bug reports in the repository are essentially randomly ordered: bug reports are opened at approximately the same time that problems, anomalies, etc., are experienced. Given the relatively large population of people opening bug reports on a complex software system, we assume that patterns in the bug reports are insignificant. Thus we don’t expect patterns to exist such that bug reports of a certain type are clustered in one range of the repository. For example, we don’t expect all of the bug reports related to Product A to be clustered in one part of the repository, and Table 3-1 shows that this is not the case. This is in contrast to alphabetized lists of names, where names of people from the same ethnic groups are likely to be clustered (e.g., the O’Briens, O’Connors, etc.) (Bailey, 1994) (p.91).

Examination of the Tables 3-1, 3-2, and 3-3 in this section (above and below) reveal some patterns. However, none of the patterns detected introduce the type of bias noted above and render the application of systematic random sampling invalid. First, there are 55 bug reports that predate, based upon the “opened date” in the bug report, the bug report with the lowest bug report number. Bug reports of this kind are here called legacy bug reports. The bug reports with earlier creation dates were imported into this repository from other pre-existing bug tracking systems. These 55 legacy bug reports were apparently deemed of sufficient import to this F/OSS development community to be migrated into the new tool so they could continue to be tracked. The second consistent pattern is that for every month since April 1998, the largest proportion of bug reports are associated with Product A, followed by bug reports associated with Product B, Product C, and then Product D. The third pattern is the rise and fall of the number of bug reports created and associated with specific products. For example, the number of bug reports created and classified as Product R bug reports declines to zero in April 1999 (only 4 bug reports are created and associated with this product after April 1999). Other products were created later, like Product G and Product N. The first bug reports for these products appear in May 2002 and June 2002, respectively.
In addition to the systematically drawn sample, other bug reports were added to the sample in order to (1) include bug reports with particularly notable characteristics in the analysis and (2) include bug reports created on products otherwise not represented in the sample. This is referred to in grounded theory as *theoretical sampling* (Strauss & Corbin, 1990).

In this report, when specific bug reports are referred to in the text, they are identified with the serial number assigned to them in the sample. For example, the first bug report in the sample is referred to as BR 1 and the last referred to as BR 385. For other bug reports theoretically sampled, a letter from the alphabet is used to designate the bug report. For example, BR-A, BR-B, BR-C, etc.

The Sampling Procedure

The sampling ratio was determined based upon the need to select 385 cases from the population of 182,864 bug reports opened between 04/07/1998 and 11/30/2002. The number 182,864 was divided by 385, yielding a sampling ratio of 1:474.971, which was rounded to 1:475.

The starting point was selected to be some distance beyond the lowest numbered bug report in the repository because it was already known that many missing bug reports existed in the lower range of the population (see discussion of the population above). Starting the selection process farther from the beginning of the population reduced the risk of selecting a missing bug report. The last bug report in the sample was opened slightly past the stated population cutoff date of 11/30/2002, a side effect of rounding the sample ratio up to 475.

The bug report numbers were recorded in a sample management spreadsheet used to coordinate and summarize further work on the sample. Other values encoded in the spreadsheet included the date the bug report was retrieved from the repository; the date the bug report was created; the date the bug report was “open coded,” the date the bug report was imported into the qualitative analysis tool (Atlas/ti), a subjective rating of how interesting the bug report is and the product the bug report is associated with.
Characteristics of the Sample

All of the bug reports in the sample exist: no missing bug reports were sampled. One bug report is not available for retrieval; it is an access-restricted bug report. If you try to retrieve it, you receive an “access denied” message: you are allowed to view this bug report only if you have an account with the appropriate access privileges. No legacy bug reports were sampled.

Bug reports continue to be added to the repository, but the population cut-off date is used to minimize the number of bug reports in the sample that are still active (that is, the underlying bugs are being diagnosed, the bug reports are having comments added, software modifications are being made, software modifications are undergoing tests, and so on) and are having changes made to them. The goal was to perform the analysis on as static a “snapshot” of the bug report repository as possible.

Changes made to bug reports after November 30, 2002 were ignored in the analysis. This may not be necessary, but avoids complications caused by looking at a bug report at different points in time and having to deal with changes in status or priority that have occurred to a particular bug report between the time the sample was first prepared and a later date when coding or analysis is being performed. The practical solution to this was to print / save all the bug report records in the sample from the most constrained time period possible. All analytic work was performed on that set of printed / saved reports. The sample selection was performed on 12/23/2002, and the preparation of the sample began with the oldest bug reports. Thus, preparation of each instance of the sample began at least 30 days after the bug report was created. This was purposely done to ensure that every bug included in the sample would have been open and active for at least 30 days, which is the time within which at least half of the bug reports in other F/OSS projects are marked resolved (Mockus, Fielding & Herbsleb, 2002) (p.338). Bug reports that remain in an open or active status for a long period of time are not excluded because the activities related to these bug reports is as interesting as the activities related
to bug reports that are closed in shorter time periods. The normative process suggests that bugs should be resolved quickly but the data suggests that there are many deviant situations, and one important metric of deviation is the length of time the passes between bug report creation and bug report resolution.

Examination of Tables 3-2 and 3-3 illustrates the fit of the systematic random sample to the population from which it was drawn. The systematic random sampling process results in a sample that matches the growth in the population of bug reports over time. In 1998, for example, there were fewer than 2,000 bug reports in the repository. The sample, appropriately, includes only a few (4) bug reports from that year. In contrast, more than 5,000 bug reports were added to the repository each month in 2002. The sample includes at least 10 bug reports from each of those months.

Table 3-2 Bug Report Sample – Distribution by Product by Month

<table>
<thead>
<tr>
<th>Month, Year</th>
<th>Product A</th>
<th>Product B</th>
<th>Product C</th>
<th>Product D</th>
<th>Other Products</th>
<th>Total Bug Reports in Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998-04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1998-05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1998-06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1998-07</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1998-08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1998-09</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1998-10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1998-11</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1998-12</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1999-01</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1999-02</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1999-03</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1999-04</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1999-05</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1999-06</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1999-07</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1999-08</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1999-09</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1999-10</td>
<td>3</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1999-11</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>1999-12</td>
<td>4</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>2000-01</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>2000-02</td>
<td>6</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2000-03</td>
<td>6</td>
<td>2</td>
<td></td>
<td>1</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>2000-04</td>
<td>6</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2000-05</td>
<td>6</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2000-06</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2000-07</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>2000-08</td>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

90
### Table 3-2, cont.

<table>
<thead>
<tr>
<th>Month, Year</th>
<th>Product A</th>
<th>Product B</th>
<th>Product C</th>
<th>Product D</th>
<th>Other Products</th>
<th>Total Bug Reports in Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-09</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2000-10</td>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2000-11</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>2000-12</td>
<td>2</td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>2001-01</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2001-02</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>2001-03</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2001-04</td>
<td>6</td>
<td>2</td>
<td></td>
<td>1</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2001-05</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>2001-06</td>
<td>9</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>2001-07</td>
<td>6</td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>2001-08</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>2001-09</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>2001-10</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>2001-11</td>
<td>6</td>
<td>3</td>
<td></td>
<td>1</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>2001-12</td>
<td>6</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>2002-01</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>2002-02</td>
<td>11</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>2002-03</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>2002-04</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>2002-05</td>
<td>9</td>
<td>2</td>
<td></td>
<td>4</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>2002-06</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>2002-07</td>
<td>8</td>
<td>2</td>
<td></td>
<td>1</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>2002-08</td>
<td>9</td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>2002-09</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>2002-10</td>
<td>12</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>2002-11</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>2002-12</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td><strong>275</strong></td>
<td><strong>71</strong></td>
<td><strong>7</strong></td>
<td><strong>9</strong></td>
<td><strong>23</strong></td>
<td><strong>385</strong></td>
</tr>
</tbody>
</table>

Table 3-3 compares the sample to the population in terms of the product value associated with each bug report. The table shows that, in terms of product representation, the sample matches the population well. Only a few products, those with fewer than 704 bug reports created before 12/01/2002, are not represented in the systematic sample. All products represented in the sample are present at a rate that is within 2.5% of their representation in the population as a whole (Product B bug reports comprise 18.4% of the sample but only 15.9% of the population).
## Table 3-3 Comparison: Sample to Population, by Product

<table>
<thead>
<tr>
<th>Product</th>
<th>Sample Count</th>
<th>Percentage of Sample</th>
<th>Population Count</th>
<th>Percentage of Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product A</td>
<td>275</td>
<td>0.71</td>
<td>128,759</td>
<td>0.7</td>
</tr>
<tr>
<td>Product B</td>
<td>71</td>
<td>0.18</td>
<td>28,797</td>
<td>0.16</td>
</tr>
<tr>
<td>Product C</td>
<td>9</td>
<td>0.02</td>
<td>4,517</td>
<td>0.02</td>
</tr>
<tr>
<td>Product D</td>
<td>7</td>
<td>0.02</td>
<td>4,819</td>
<td>0.03</td>
</tr>
<tr>
<td>Product E</td>
<td>5</td>
<td>0.01</td>
<td>2,829</td>
<td>0.02</td>
</tr>
<tr>
<td>Product F</td>
<td>3</td>
<td>0.01</td>
<td>692</td>
<td>0.0</td>
</tr>
<tr>
<td>Product G</td>
<td>3</td>
<td>0.01</td>
<td>2,220</td>
<td>0.01</td>
</tr>
<tr>
<td>Product H</td>
<td>2</td>
<td>0.01</td>
<td>122</td>
<td>0.0</td>
</tr>
<tr>
<td>Product I</td>
<td>2</td>
<td>0.01</td>
<td>462</td>
<td>0.0</td>
</tr>
<tr>
<td>Product J</td>
<td>2</td>
<td>0.01</td>
<td>2,098</td>
<td>0.01</td>
</tr>
<tr>
<td>Product K</td>
<td>2</td>
<td>0.01</td>
<td>1,370</td>
<td>0.01</td>
</tr>
<tr>
<td>Product L</td>
<td>1</td>
<td>0.0</td>
<td>688</td>
<td>0.0</td>
</tr>
<tr>
<td>Product M</td>
<td>1</td>
<td>0.0</td>
<td>375</td>
<td>0.0</td>
</tr>
<tr>
<td>Product N</td>
<td>1</td>
<td>0.0</td>
<td>1,065</td>
<td>0.01</td>
</tr>
<tr>
<td>Product O</td>
<td>0</td>
<td>0.0</td>
<td>23</td>
<td>0.0</td>
</tr>
<tr>
<td>Product P</td>
<td>0</td>
<td>0.0</td>
<td>522</td>
<td>0.0</td>
</tr>
<tr>
<td>Product Q</td>
<td>0</td>
<td>0.0</td>
<td>41</td>
<td>0.0</td>
</tr>
<tr>
<td>Product R</td>
<td>0</td>
<td>0.0</td>
<td>476</td>
<td>0.0</td>
</tr>
<tr>
<td>Product S</td>
<td>0</td>
<td>0.0</td>
<td>703</td>
<td>0.0</td>
</tr>
<tr>
<td>Product T</td>
<td>0</td>
<td>0.0</td>
<td>177</td>
<td>0.0</td>
</tr>
<tr>
<td>Product U</td>
<td>0</td>
<td>0.0</td>
<td>611</td>
<td>0.0</td>
</tr>
<tr>
<td>Product V</td>
<td>0</td>
<td>0.0</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Product W</td>
<td>1</td>
<td>0.0</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Product X</td>
<td>0</td>
<td>0.0</td>
<td>55</td>
<td>0.0</td>
</tr>
<tr>
<td>Totals</td>
<td>385</td>
<td>1.0</td>
<td>181,366</td>
<td>1.0</td>
</tr>
</tbody>
</table>

### Data Collection and Preparation

As mentioned in the section on sampling, I wanted to create versions of the bug reports in the sample that were as consistent as possible in terms of the time they were collected from the repository and prepared for analysis. This was a concern because preliminary analysis of bug reports indicated that bug reports, even those that had been marked “resolved,” could be modified long after resolution. One example of this kind of change is the later change of the product name for a group of bug reports. For example, BR 1 in the sample had its product attribute changed more than a year after it was closed (from Product A to Product C) as part of a mass change to all Product C-related bug reports.
In practice, this was difficult because of the relatively large sample size and the time it took to look up and retrieve copies of each bug report in the sample. The retrieval process took place between 12/23/2002 and 01/30/2003, a period of about 5 ½ weeks.

I created and used a sample management spreadsheet to organize the sample preparation work. For each bug report in the sample, I used the Web-based repository interface to retrieve each bug report by using its bug report serial number. Using a popular Web browser, I would take the following steps for each bug report in the sample:

1. Save an HTML version of the full bug report
2. Save an HTML version of the bug report “formatted for printing”
3. Save an HTML version of the “bug activity” log
4. Save a text version of the bug report “formatted for printing”
5. Save a text version of the “bug activity” log

The text versions were created from the saved HTML versions rather than by returning to the live bug report repository to ensure consistency between the HTML and text versions of the same bug report.

Following collection of all bug reports in the systematic sample, I imported the text version of the bug report “formatted for printing” into a qualitative analysis tool called Atlas/ti. Each of these 385 text format bug reports was a document instance that underwent content analysis using the grounded theory procedures described below.

Data Analysis

Coding

Open coding began on a subset of 40 bug reports (the earliest 40 bug reports in the sample, beginning with BR 1) between 12/25/2002 and 12/31/2002. This initial coding pass was done using printed copies of the bug reports instead of the qualitative
analysis tool, Atlas/ti. (I had not made a decision on whether to use a computer-based tool or not, nor had I decided which tool would be most appropriate. I was also unsure of what kinds of information, themes and ideas might emerge from coding the bug reports, so I wanted to be as open and flexible as possible as I read and coded the first few bug reports.)

The goal of this coding pass was to begin development of a grounded set of codes to be applied to the open coding of the entire sample. I did not expect to create a complete set of codes from this small “sample within the sample,” but I did find a number of important, recurring themes in this small set of bug reports.

Open coding is a process whereby you “…nam[e] and categoriz[e] phenomena through close examination of data” (from Strauss & Corbin, 1990) (p.62). Open coding is the first analytic step in grounded theory. As I read the bug reports, I looked for the phenomena – forms of information, ideas, activities, processes, artifacts, events, social interactions – represented within the bug reports and assigned names to each of them. As the process continued, and more concepts were identified, the concepts were grouped into categories. The names assigned to concepts and categories were in some cases borrowed from other research or theories, but the language used by members of this F/OSS development community suggested most of them.

Properties and dimensions of individual categories also emerged during open coding. The properties, and sub-properties, of a category are the attributes that comprise the category. Open coding can be applied at different levels of granularity: at the word or phrase level, at the comment level, or at the bug report level (Strauss & Corbin, 1990, Chapter 5).

After the selection of Atlas/ti as the tool for supporting the coding and analysis work, I began open coding of the entire sample. I returned to the beginning of the systematic sample (BR 1) in order to (1) encode the previously coded bug reports in Atlas/ti and (2) apply the revised codes as defined in the codebook.

There were some differences between how I coded the same bug reports during the initial coding pass and the second pass. For example, I decided to code a few key
parts of the structured bug report: the product, status, summary and resolution. This would allow me to generate certain basic counts within the systematic sample. There were also differences in how I applied codes during the two coding passes. During the first coding pass I tended to apply a single code to entire comments or paragraphs occurring within comments. During the second pass I was likely to apply multiple codes to individual comments resulting in finer-grained coding. In many cases, the same text may be coded with more than one code. In some cases, multiple codes are applied to the exact same text string, while in other cases the text strings may overlap.

During open coding I also kept track of additional features and phenomena found in each of the bug reports. I noted examples of the following:

**Bug Report Type:** The bug type labels are often drawn from the language used by this F/OSS development community itself. Examples are “Regression,” “Maintenance,” “Change Notice,” etc. In most cases, these designations seem to designate a quality of a bug report and imply particular categorization outside of the categories provided for by the structured repository fields like “product.” One particular example of this is the “Meta” bug report type, explained more fully in Chapters Four and Five.

In other cases, these types refer to the context in which the underlying bug was reported. Examples of this are “Smoketest” and “Regression.” Both labels are used within the community, but are overlapping designations. A typical regression bug occurs when software – comprising modules A, B and C – is tested at time 1; module C is modified and tested; the software is tested again at time 2. If a problem occurs in either module A or B at time 2 that was not present when the software was tested at time 1, a regression bug is identified. In this context, the term regression implies that the software, for whatever reason, has “regressed,” or become less useful than it was before. The term smoketest is used within this
F/OSS development community to indicate standard regression tests run daily on new software builds.

**Trajectories:** The bug report comments and the bug activity data form a representation of a bug report’s trajectory – a path, progression, or line of development resembling a physical trajectory – from the time the bug report is created until it is set to a terminal state (e.g., “status = verified”). But the bug report also reveals information about how the work of managing software problems is aligned; who participates in that process; how the organization of the community affects and is affected by the information contained in the bug report. The term trajectory comes from the work of Anselm Strauss et. al. (1985) on the ways in which medical work is organized.

**Dialogs / Negotiations:** Summarization of the negotiation threads identified in individual bug reports. These notes organize the individually coded instances of negotiation activity into discrete threads that often span multiple and non-adjacent comments. Individual comments may include statements that refer to more than one active negotiation thread. Chapter Seven contains a detailed analysis of negotiation in bug reports.

**Timing:** A description of the temporal rhythm evident within a bug report.

**Bug Report Networks:** A description of the various relationships between the bug report being coded and other bug reports. These relationships may be formalized as particular relationships like “duplicate,” “depends on,” or may be informal references (e.g., “see also” references). See Chapter Five for a detailed discussion of bug report networks.
Establishing Trustworthiness of Results

Validity

Coding of the bug report data will be performed using grounded-theory procedures. Grounded-theory seeks to generate concepts and categories empirically from data. Face validity is assured through the use of grounded theory as the analysis process seeks to map data directly to the concepts drawn from the data (Bailey, 1994) (p. 68-72).

External validity is established by comparing the results of this analysis to other research studies examining the same or similar phenomena (Strauss & Corbin, 1990) (p.52). In this project, other research with which this project was compared included research on negotiation in both different and similar domains (Strauss, 1978; Egger & Wagner, 1992); theories of cooperation and coordination in various domains (Schmidt & Simone, 1996; Crowston, 1997); existing descriptions of the software problem management process (Carstensen, Sørensen & Tuikka, 1995; Schmidt & Simone, 1996; Crowston, 1997; Kajko-Mattsson, 1999); sensemaking (Weick, 1995); social order (Strauss, 1978; Fitzpatrick, Tolone, & Kaplan, 1995); and problem solving and debugging in computing work (Gugerty & Olsen, 1986; Flor & Hutchins, 1991; Spinuzzi, 2001). The external validity of this project can also be assessed in the future by conducting some or all of the related follow-up studies recommended in Chapter Nine.

Reliability

Reliability is the consistency of a measurement over time. In the context of this study, coder reliability is the main concern, specifically intra-coder reliability because the same individual has performed all of the data coding. Construction and use of a coding procedure and codebook minimized intra-coder reliability problems.
Generalizability

One of the goals of information studies research is to generate explanations and theory that can be applied in a variety of settings. The best means of ensuring generalizability of results is probably the careful replication of a study in other settings and sets of conditions. Such replication is beyond the scope of this dissertation project. However, paying attention to issues of external validity (discussed above) improves generalizability to some degree.

The striking similarities between my own personal experience as a worker in several different software development and information systems infrastructure management organizations and the SWPM process employed in this community also indicate to me that these results are largely applicable to other settings. I have worked in a very large organization that used a paper-based “trouble ticket” system that was replaced by a complex, homegrown relational database system, which was in turn replaced by a much more complex commercial relational database system. This quasi-governmental organization performed a mix of distributed infrastructure design and management software development in the financial sector. I’ve also worked in a medium-sized privately held corporation in the health services field as a member of a small software development team where several simple bug report repositories were used (including a paper-based system in use when I joined the organization). I’ve also done software development in an academic setting where no problem management system of any kind was in use.

Advantages and Limitations of Methods Used

Advantages

The method described here has some characteristics that make it effective in the context of this study. One of the objectives of this project was to create a map of the
distributed work done in this community in support of managing software problems. Since no previous work had established the kind of complete description of SWPM work provided here, qualitative methods, particularly grounded theory, was an appropriate choice. Reliance on document analysis was also an appropriate choice given the distributed nature of the F/OSS development community studied here.

**Limitations**

The method described here has some characteristics that limit its usefulness as a research approach in this context. This project relies exclusively upon the analysis of documents (bug reports, community process documentation, and related research). There are two key implications of this. First, there are other types of repositories and digital documents besides bug reports that could provide additional insights into this community’s SWPM process. These include e-mail archives, chat archives, newsgroup archives, and the comments embedded in the code created by the community to fix these software problems. Second, there are some kinds of information that are impossible to glean from the study of documents. It isn’t possible to ascertain from the bug reports alone the thought, decision-making processes, and activities employed by community members that are not included in the bug reports. One particular issue of this type concerns the perceived utility and actual utilization of bug report networks in support of managing software problems (discussed further in Chapters Five and Nine). This limitation could be ameliorated by conducting interviews with participants in the SWPM process as they are managing software problems and bug reports.

As noted above, intra-coder reliability is a risk because I am the only coder of data for this project. It is possible that my approach to coding the bug report data may either be inconsistent in a non-systematic way or change systematically as iterations of bug report coding and analysis occur over time.
CHAPTER FOUR: SOFTWARE PROBLEM MANAGEMENT
COMPONENTS

Introduction

This chapter summarizes the various possible units of analysis – the information, activities, and concepts – revealed by the analysis of a random, systematic sample of bug reports drawn from this F/OSS development community’s bug report repository. The units of analysis found in this bug report repository are described in order, the most general unit of analysis first. These units of analysis are society at large, F/OSS meta communities, F/OSS communities, bug report repositories, bug report networks (BRNs), bug reports, bug report state sequences, dialogs (such as question and answer sequences and instances of negotiation), comments, and statements. More thorough analysis selected concepts, activities, and information follows in chapters Five through Eight. Together, these five chapters document the mapping of software problem management (SWPM) that is one of the fundamental objectives of this study.

The Units of Analysis

This section provides an overview of the analytic units that have been identified through the close examination of the F/OSS bug report repository studied during the course of this project. This section establishes definitions of important structural features, first-class objects, and virtual objects represented in both the repository and the bug reports, contributing to the conceptual map of SWPM. Analytic units that are a primary focus for this project are clearly identified. The analytic units are presented in order from the largest, or most general, to the smallest, or most detailed (see Figure 4-1).
Society At Large

The most general unit of analysis is human society at large. Free and open source software projects have an effect on anyone who makes direct or indirect use of computing infrastructure. For example, 36% of public servers on the Internet use Linux or FreeBSD (Netcraft.com, 2001). 64% (about 25,800,000) of the World Wide Web servers use software from the Apache project (Netcraft.com, 2003). Thus it is unlikely any user of the Internet has not used free / open source software. Also, as noted in the Introduction chapter, the quality of software is a general social concern because software is embedded in almost all modern infrastructure, objects, and tools. Effects of SWPM and interactions between this F/OSS community and human society are beyond the
scope of this project and are not considered further. Society at this scale is certainly affected by software whether the software is created using traditional, closed source or F/OSS approaches. However, issues of information, activity, and social order at this level of analysis are beyond the scope of this project.

F/OSS Meta-Communities

Specific F/OSS meta-communities, like SourceForge (SourceForge.net, 2004), Tigris (Tigris.org, 2004), and Apache (Apache.org, 2004) serve as collections of individual F/OSS projects. These meta-communities exist to provide and organize support for the wider F/OSS community. SourceForge, for example, provides “a centralized place for Open Source developers to control and manage Open Source software development” for “tens of thousands of projects” (SourceForge.net, 2004). Tigris describes itself more narrowly as “a home for open source software engineering tool projects” (Tigris.org, 2004). Apache.org, best known for producing the most widely used Web server in the world, also plays host to a number of (about 15 as of this writing) other software projects (Apache.org, 2004). These meta-communities of F/OSS communities are interesting in and of themselves, but are not discussed further here.

F/OSS Communities

Specific F/OSS communities are at the next analytic level. While these communities are often characterized as consisting only of programmers and engineers who are also the users of the software, they also include people who play less technical roles in each community. The Mozilla community offers a good example of a project with a variegated constituency. While the Mozilla browser may be more popular among programmers than other users, many people who do not participate in the design, coding, and production of the system use it. Some users may occasionally report bugs by either using the bug report repository directly or by having browser crash data sent
automatically to Mozilla.org. The core of this community – the designers, developers and quality assurance workers – is most visible in this project because of their direct participation in software problem management.

Each F/OSS community shares a set of tools that constitute some of the critical infrastructure for a project. Typical tools include compilers, version control systems, operating systems, documentation and bug report repositories. This project focuses on the bug report repository with one F/OSS development community.

**Bug Report Repositories**

The bug report repository is one of the first-class objects in the hierarchy of analytic units and a critical piece of infrastructure for any non-trivial software development project. There are at least three ways for repositories of this type to be employed, depending on the kinds of system and software problems the organization needs to manage, and depending upon how the organization wishes to manage problem information. The exact technical characteristics of the systems by themselves do not determine how it is used in a given organizational setting; the unique conventions and processes developed by each community are the primary determinants of how each system is deployed.

*Help desk systems* are intended to be easy for end users to use and tend to have simple interfaces for submitting problems. One goal of this kind of system is support of *case-based* retrieval of information:

Episodic, case-based systems … aim … to capture knowledge about an instance or "episode" of problem-solving and to support people in re-applying the knowledge in other, similar or analogous episodes. Examples include helpdesk systems, FAQs, and issue-tracking systems when they are used as repositories that allow users to see whether issues have already been stated or solved. (Gasser & Penne, 2002)

Help desk support personnel are interested in creating and maintaining standard responses to common problems. They want a system that makes retrieval of existing
relevant cases simple, based on symptoms and problem characteristics. Ideally, retrieval of a relevant case makes recovery from the problem trivial.

The Chandra Users Mail Archive, for example, uses an e-mail interface for collecting and threading submissions and responses to submissions. The Chandra Helpdesk is a relatively simple system (each record in this repository has fifteen formatted fields plus free-text comments and the ability to handle attachments). It is easier to understand and use than Bugzilla, which is designed for use primarily by the members of a software development organization.

*Software problem management systems,* or *bug report repositories* (the term used in this document), the second type of system, are designed to support the management of bugs related to one or more software systems. The goal of this kind of system is somewhat different from the goal of a help desk system. Instead of a focus on recording and recovery of episodic information, these systems support *sensemaking* (Weick, 1995). These systems:

… accumulate information and knowledge in order to refine an understanding of a situation. These systems track the development of knowledge about a situation, and they also represent knowledge about the process of addressing the situation: where in the process trajectory are we? What is the division of labor and responsibility? … in that it supports [the continuous] effort of assembling knowledge resources and work processes into a set of conclusions about the nature of bugs and how to repair them. (Gasser & Penne, 2002)

The F/OSS development community whose SWPM process is studied in this project uses a sensemaking problem management system. The repository used by this community, Bugzilla, has more than twenty structured fields as well as alternate views of bug reports (e.g., the activity log: see Figures 5.3, 5.4, and 5.5 below; bug report dependency displays, etc.) In contrast, the Chandra Helpdesk, by billing itself has a Helpdesk system rather than a bug reporting or software problem reporting system is more likely to be utilized by end-users to ask “how do I…” kinds of questions than
Bugzilla. In the community studied in this project, “how do I…” questions are directed to other forums (from bug 375):

------ Additional Comment #1 From <Commentator_1> 2002-11-03 10:23 ------
<reporter>, strictly speaking, you're asking for technical support. This database is only intended for bug reporting. You should ask this question in a forum intended for end-user support.

The third type of system, general problem management systems, combines elements of episodic, case-based, and sensemaking tracking systems. They are intended to support the management of a variety of types of problems including recurring problems, “how do I” questions from users, and software defects (bugs). Case-based retrieval is important in this kind of system, but the system is designed to be sophisticated enough to track non-recurring problems, like software problems, as well as track duplicates, related software changes, etc. IBM's InfoMan and many home-grown systems fall into this category. Communities and organizations that have responsibility for a wide variety of systems and functions often use this kind of system. Organizations involved in both real-time supervisory control and design work, where both recurring problems supported by episodic, case-based systems and software problems requiring a sensemaking style system occur, have a need for a system of this complexity (Sandusky, 2003).

Any of the three kinds of systems described above can be implemented in a number of ways. The system can be paper-based: someone fills out a form when a problem is identified and a process exists to route the paper to the people who have the ability to correct the problem (Carstensen, Sørensen & Tuikka, 1995). Threaded e-mail archives can be used to report and track problems (Chandra X-Ray Center. Chandra Users Mail Archive: By Thread). Database systems with simpler database and software designs can also be used to manage software problems (Chandra X-Ray Center. Welcome to the CXC Helpdesk). In most cases, however, communities and organizations use complex database-driven systems (e.g., Scarab, Bugzilla, IBM's InfoMan; home-grown systems), providing distributed access to problem data and tools for problem management. Database-driven systems have been used since at least the early 1980s. Bugzilla is an
example of a successful F/OSS bug report repository (Bugzilla.org, 2004) specifically geared toward the management of “software defects” and using a public, Web-based interface. Bugzilla is implemented as a set of scripts, or programs, utilizing a relational database management system for data management and access.

**Bug Report Networks**

Now I will describe analytic units found within the bug report repository studied in this project in sequence, starting from the largest, or most general, to the smallest, or most detailed. Community members create bug report networks (BRNs) by progressively asserting various formal and informal relationships between bug reports (BRs). BRNs are a new type of information object discovered and described here for the first time. BRNs are a virtual object in this community’s bug report repository. The following formal relationships may be established in Bugzilla. These formal relationships are established by making an explicit association between bug reports in the repository.

**Duplicate**: bug reports with this relationship should identify one bug report that is the “master” bug report with one or more “duplicate” bug reports. This is expressed in the database as a formal, bi-directional relationship: from a “master,” you can find all identified duplicate bug reports; from a duplicate, you can find its “master” bug report.

**Blocks**: a bug report is identified as “blocking” if it is holding up work on part of the code or another bug report.

**Depends On**: “If a bug can’t be fixed until another bug is fixed, that’s a dependency. For any bug, you can list the bugs it depends on and bugs that depend on it” (Mozilla.org, “bugs”). Or, “If this bug cannot be fixed unless other
bugs are fixed (depends on), or this bug stops other bugs being fixed (blocks)”
(Mugzilla.org, “A Bug’s Life Cycle”).

Community members also make informal associations, usually within the
comments that are appended to the bug report. For example:

- This looks related to #X (BR 9)
- See comments on X -- same applies here I think. (BR 21)
- My fix for X kinda helps fixing this too. (BR21)
- Could be dup of bug X or bug Y. (BR 35)
- Should bug X be added to this? (BR 220)
- in bug X comment 1, I said this works in 4xp. (BR 265)

More analysis of BRNs found in this community’s bug report repository is
presented in the next chapter.

Bug Reports

Bug reports are a first-class object in the hierarchy of analytic units shown in
Figure 4-1. Bug reports are also the primary unit of analysis in this project. In this section,
the anatomy of the bug reports is defined, based upon the characteristics of bug reports
held within a repository of the type used in the F/OSS community studied here
(Bugzilla). This description is generic and applies to any implementation of Bugzilla. Each
community employing Bugzilla can make selected standard customizations, or, because
the community has access to the Bugzilla source code, more extensive customizations.

The Bugzilla repository is a relational database that holds bug reports. Each bug
report has a consistent structure comprised of two main sections: a set of structured
attributes, many of which have constrained sets of legal values (drawn from controlled
vocabularies), a set of unstructured text fields (the bug description and comments) and optional attachments.

Figure 4-2 shows the text representation of a simple bug report drawn from the repository investigated in this project. This sample bug report provides a sense of the information that a bug report usually contains. In this case, the reporter and the assignee are the same individual (shown as <Reporter/Assignee> in the sample report below). Identities of individuals and some system-related names are disguised (framed by angle brackets: thus Microsoft Windows becomes <Commercial Operating System>). The line numbers in parentheses at the beginning of each line were added to facilitate the discussion in the text.

(1) Bug 140 - Paste is enabled for readonly textfields
(2) Bug#: 140   Product: <product_name> Version: Trunk
(3) Platform: All   OS/Version: All   Status: VERIFIED
(4) Severity: normal   Priority: --   Resolution: FIXED
(5) Assigned To: <Reporter/Assignee>   Reported By: <Reporter/Assignee>
(6) QA Contact: <QAContact_1>   Component: Editor: Core   Target Milestone: ---
(7) URL:   Summary: Paste is enabled for readonly textfields
(8) Keywords:   Status Whiteboard:   Opened: 2001-01-24
(9) Description:
(10) Paste is enabled in the context menu for readonly textfields (but it still doesn't do (11) anything). Patch coming; <CC_1>, can you <perform second level code review>?
(12) -------- Additional Comment #1 From <Reporter/Assignee> 2001-01-24 21:22 --------
(13) Created an attachment (id=23433) [patch] disable paste if not modifiable
(14) -------- Additional Comment #2 From <CC_1> 2001-01-25 10:51 --------
(15) <second level code review> <CC_1>
(16) -------- Additional Comment #3 From <Commentator_A> 2001-01-25 12:34 --------
(17) <first level code review> <Commentator_A>
(18) -------- Additional Comment #4 From <Reporter/Assignee> 2001-01-25 13:36 --------
(19) checked in.
(20) Status changed from New to Resolved
(21) QAContact changed from QAContact_1,QAContact_2 to QAContact_1
(22) Resolution changed from NULL to Fixed
(23) -------- Additional Comment #5 From <QAContact_1> 2001-02-07 13:43 --------
(24) verified in 2/6 build.
(25) Status changed from Resolved to Verified

Figure 4-2 Anatomy of a Bug Report
Lines 1-8 hold the structured portion of the bug report. There are several important elements contained within this portion of the bug report.

Line 1
- Shows the bug report’s serial number and the value of the “summary,” which is repeated in line 7.

Line 2:
- Repeats the bug report’s serial number
- Shows the product this bug report has been entered against (This community is involved in producing more than 20 different “products.”); controlled vocabulary.
- Shows the version of the software (in this case, the bug is reported against the “trunk” of the code tree). This value may be drawn from these values: 1.0 Branch; 1.4 Branch; Other Branch; Trunk; controlled vocabulary.

Line 3
- Shows the computing platform this bug report is entered against (In this community, choices include PC, Macintosh, All, Other, various Unix platforms.); controlled vocabulary.
- Shows the specific OS version associated with the bug report (a list of various PC, Macintosh, and platforms usually associated with Unix/Linux variants); controlled vocabulary.
- Shows the status of the bug report (VERIFIED) drawn from these values: UNCONFIRMED; NEW; ASSIGNED; REOPENED; RESOLVED; VERIFIED; controlled vocabulary.
Line 4

- Shows the bug report’s severity (normal) drawn from an ordered list (most severe listed first) of blocker; critical; major; normal; minor; trivial; enhancement; controlled vocabulary.
- Shows the bug report’s priority; and its priority (--, or unset; P1; P2; P3; P4; P5 are the ordered values, with P1 indicating “most important”); controlled vocabulary.
- Shows the bug resolution value (FIXED) drawn from these values: FIXED; INVALID; WONTFIX; DUPLICATE; WORKSFORME; MOVED; controlled vocabulary.

Line 5

- Shows the e-mail address of the person to whom the bug report is assigned. This value indicates who currently is in the role of the Assignee, the person responsible for fixing this problem.
- Shows the e-mail address of the person who created the bug report. This value indicates who currently is in the role of the Reporter, the person who reported this problem.

Line 6

- Shows the e-mail address of the person to who is the designated QAContact. This value indicates who currently is in the role of the QAContact, the person responsible for verifying that this problem is fixed, or the bug report can be marked as having its resolution verified.
- Shows the component within the selected product this bug report is believed to be associated with; drawn from a large but limited list; the controlled vocabulary available in this field depends on the value of the product field for this bug report; controlled vocabulary.
- Shows the release milestone for which resolution of the underlying bug is projected; drawn from a large but limited list of values; controlled vocabulary.

Line 7
- Shows an optional Web address (URL) that is relevant to the bug report.
- Shows the short summary description of the bug report (repeated in line 1)

Line 8
- Shows the optional keywords that have been associated with this bug report.
- Shows the special area called the status whiteboard, used to communicate particular kinds of information to the community about this bug report.
- Shows the date this bug report was created and added to the repository.

The less structured portion of the bug report begins on line 9 and ends on line 25. Line 9 is the heading for the Description area. Lines 10-11 contain the description of the problem itself, which the reporter is required to supply when creating a new bug report. This example shows some of the typical information for this section. Text describing a procedure to recreate, or reproduce, the bug is missing from this bug report. Steps to reproduce the behavior are usually included in order to help the person confirming the problem or trying to fix the bug to observe the problem behavior.

Information about the version of the software in use (the specific system build in use when the problem was found) is another typical kind of information missing from this bug report. The example below contains descriptive information including system build and steps to recreate the problem (from bug report 193):

From Bugzilla Helper:
<system identifiers removed>
BuildID: 2001063008
<system> freezes totally.
Works fine in <other systems>

Reproducible: Always
Steps to Reproduce:
1. Go to <location>
2. Click "Reference Manual"
3. Enjoy

Actual Results: <system> freezes completely

Expected Results: Open the <reference manual>

I also tried this with my work machine, which has NT4 and slightly newer build.

Comments

Comments are a structural sub-component of a bug report and are first-class objects in the repository (comments can be identified and extracted as entities from the database). Lines 12-25 contain comments added to the bug report after the reporter has created it along with indications of modifications to the fixed fields by commentators (e.g., lines 20, 21, 22 and 25). Each comment has the following structure:

The comment header: consists of the static values “------- Additional Comment #” at the beginning of the first line and “-------“ at the end of the line (see lines 12, 14, 16, 18, and 23). The serial number of the comment is added after the octothorpe (#). The identity of the community member making the comment is added as “From John Doe” or as “From jdoe@some_place.com,” depending on how the user’s repository profile is defined. A timestamp is concatenated after the community member’s identity.

Text: what follows the comment header is primarily text entered by the community member. In some cases, modifications of the fixed fields appear (lines 20-22, 25), with or without comment (these changes are presented by Bugzilla on a separate Web page (see Figures 5-4 and 5-5); they have been included in this text-only representation to provide a unified view to support this discussion).
**Attachment:** If a community member creates an attachment, Bugzilla appends a notification like that shown on line 13. Attachments can represent code for a patch, screen shots, stack traces, or other relevant information.

Another common occurrence within comment text is the automatic insertion of a notification when a duplicate relationship is formally asserted between two bug reports. When a duplicate is formally asserted, each bug report has a comment like the following appended:

In bug report X:

*** This bug has been marked as a duplicate of Y. ***

In bug report Y:

*** Bug X has been marked as a duplicate of this bug. ***

Figures 4-3, 4-4, and 4-5 show what a bug report looks like when rendered in a Web browser. This is the representation normally used by members of the community studied in this project. The top portion of this Figure 4-3 eight different ways a bug report can be classified using some sort of controlled vocabulary:

Figure 4-3 shows one additional area that the text representation (above) does not: the CC: (carbon copy) list. These are the identities of the community members who are “carbon copied” on e-mails generated whenever the bug report changes.

Other community members may post additional comments on the bug report even if they are not otherwise involved in the management of the software problem. These participants are referred to in this report as *commentators*. 
Statements

Statements are a virtual entity within a bug report and are defined here as text within one comment within one bug report, where the beginning and end of the statement are delimited by topical consistency. What I mean is that whatever text is (subjectively) determined to be about a single topic is considered a statement. A statement can consist of multiple sentences or paragraphs of text or alternatively consist of sentence fragments or symbols. On the other hand, a comment can contain multiple statements from the same individual.

Here is an example of a comment that includes multiple statements. Paragraphs 1 and 3 are concerned with the technical merits of one design approach. Paragraph 2 is about proper behavior in handling of bug reports and is in response to the previous commentator’s contribution to the bug report which included the reaction “it’s ridiculous.” Paragraph 2 is, in effect, a comment on the way this particular bug report is being handled by the community (from bug report 374).

-------- Additional Comment #<n> From <Commentator_2> 2002-11-01 10:19 --------
Loading <component> via style attributes introduces one more code path that needs careful handling from security perspective. The more of these entry points we have, the more likely it is we will make a mistake somewhere.

If a feature is not necessary, and can easily be done some other way, we should at least think of the issues at hand before throwing “it’s ridiculous” arguments without any thoughts to the issues.

Nobody is saying at this point that we should definitely disable <component> from style attributes. But we do want to evaluate the pros and cons.

Here is another example from the same bug report of a comment, from the bug report’s QAContact, containing multiple statements. The first, one-word sentence is status information. The second sentence is rationale supporting assignment of a resolution of INVALID to this bug report:

-------- Additional Comment #<n> From <QAContact> 2002-11-01 11:23 --------
VERIFIED. There's nothing special about the style attribute here.
This final example shows a comment consisting of a single statement, expressing the opinion that the bug represented by this bug report was not caused by other changes to the system (that is, this is not a regression bug) (from bug report 201):

------- Additional Comment #2 From <Assignee_2> 2001-08-15 14:50 -------
Don't think any editor change caused this.

The following are examples of use of symbolic statements. The first means “marking as resolved, WORKSFORME” (from bug report 304) and the second means “this bug report has been assigned to the default assignee for this product” (from bug report 25):

-> wfm
-> default assignee

<table>
<thead>
<tr>
<th>Statement</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best Practice</strong></td>
<td>You don't free &lt;x&gt;IsSupports objects with &lt;x&gt;CRT::free. Please look at my changes to learn how to use &lt;x&gt;COMPtr (BR 28)</td>
</tr>
<tr>
<td></td>
<td>Also, if you note a bug as not fixed, you should reopen it, because otherwise it won't be looked at again. (BR 38)</td>
</tr>
<tr>
<td></td>
<td>For future reference, bugs should only be marked fixed when a checkin has been made to explicitly fix it. This one sounds more like wfm. No need to change it now though. (BR 110)</td>
</tr>
<tr>
<td></td>
<td>It's better to confirm if you can reproduce the problem on your own machine rather than relying on a picture of what it looks like on the reporters screen. There might be something corrupt on the reporters machine so that the problem is occuring only for him and nobody else. An independent confirmation would help to rule that out. (BR 303)</td>
</tr>
<tr>
<td><strong>Bug Analysis</strong></td>
<td>This is what seems to be going on...</td>
</tr>
<tr>
<td></td>
<td>1. &lt;path_1&gt; is passed into the &lt;x&gt;FilePath constructor.</td>
</tr>
<tr>
<td></td>
<td>2. &lt;x&gt;FileSpecHelpers::UnixToNative(...) is called and removes* the initial '/' and converts all slashes...</td>
</tr>
<tr>
<td></td>
<td>3. &lt;x&gt;FileSpecHelpers::Cannonify(...) is called.</td>
</tr>
<tr>
<td></td>
<td>However, the path is now &lt;path_2&gt; so it assumes it is a relative path and prepends Z:\DIST\WIN32_D.OBJ\BIN to the path :-(</td>
</tr>
<tr>
<td></td>
<td>4. Now things are <em>really</em> messed up! (BR 32)</td>
</tr>
<tr>
<td>Statement</td>
<td>Examples</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>Bug Analysis</td>
<td>• So, let me explain what is happening in the case of multiple new messages based on this call stack. CleanupRunning&lt;Y&gt; is called for the first copied message. What this does is call OnStopRequest which tells the first message that it is finished. The first message then finishes and determines that there's another message to copy. So, it starts the copy. Starting the copy makes the &lt;P&gt;Service call DisplayMessage. This then creates the new &lt;Y&gt; for the next copy and sets up state in &lt;P&gt;Protocol. That's where it goes wrong. All of the initialization for the next &lt;Y&gt; gets set up in the cleaning up code for the first &lt;Y&gt;. When the stack unravels, the protocol then clears out all of its values thinking that it's cleaning out the original &lt;Y&gt;'s values but in fact it's cleaning out the values of the 2nd &lt;Y&gt; and therefore the 2nd &lt;Y&gt; never finishes running and we never copy future messages. We have other bugs where cancelling 2 messages in a row don't work and where printing a &lt;P&gt; message doesn't work. I'm wondering if these are related. (BR 106) • Can you also find out if this is a regression.. if so.. what day it regressed. Seems to work on the trunk.. but the branch is crashed.. after the printout came out. (BR 124) • could this be that incremental reflow.. null pointer type bug you fixed for some &lt;T&gt; things. This sounds awfully familiar. (BR 124) • Does this sound like that one bug that &lt;name&gt; fixed a few weeks ago? (BR 124) • The crash seems to be caused by having a &lt;T&gt; which is split across two pages when printing. If you reduce the number of items in either &lt;T&gt; in the reduced test case it stops crashing. (BR 124)</td>
</tr>
<tr>
<td>Statement</td>
<td>Examples</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Bug Analysis      | • Here's what I saw: Used 1/12 build to forward a message to <name> which contained a <document> He tells me the document is corrupt I move the document from <A> to <B>, so I can see the complete bits of the message. Once visible in <alternate format>, the MIME part containing the <document> has only the MPOD "will be downloaded on demand" string. (BR 139) • Ok I investigated this a little. Here's what I found: All the checkboxes have an onclick handler that calls this function: function CheckCheckAll() { var TotalBoxes = 0; var TotalOn = 0; for (var i=0;i<document.<name>.elements.length;i++) { var e = document.<name>.elements[i]; if ((e.name != 'allbox') && (e.type=='checkbox')) { TotalBoxes++; if (e.checked) { TotalOn++; } } } if (TotalBoxes==TotalOn) {document.<name>.allbox.checked=true;} else {document.<name>.allbox.checked=false;} } So I thought, "what the hell, they are looking through my 200 new messages each time I click on their checkbox??". I then deleted all my messages but two, and guess what, the checkboxes were selected directly, no freeze or hang. So this function is really the culprit here. Maybe we are very slow on some kind of <structure> function that is included there? Anyway this isn't as bad as it seemed, it's just stupid from MS to do this (to gain what anyway?). Hope this helps, <commentator identity removed> (BR 146)
Figure 4-3 Anatomy of a Bugzilla Bug Report: Structured Fields and Input Areas  
(Drawn from Bugzilla Development Site)
Figure 4-4 Anatomy of a Bugzilla Bug Report: Comment Area

Figure 4-5 Anatomy of a Bugzilla Bug Report: Activity Log
Bug Report State Sequences

Bug report state sequences are a formal representation of bug report’s path, or progression, as represented by a limited set of formal states, from the time the bug report is created until it is set to a terminal state (e.g., “status = verified”). Bug report state sequences are based on the ordered set of values contained in two bug report fixed fields: status and resolution.

Bug report state sequences, an innovation in bug report analysis\(^7\), have been created from the sample drawn from the Bugzilla repository studied in this project. Most of these state sequences were generated automatically by querying a static database snapshot and then arranging the data in time sequence. The others were hand-generated by reading the bug reports.

Table 4-2 Bug Report Status and Resolution Values

<table>
<thead>
<tr>
<th>State Sequence Code</th>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status:</td>
<td></td>
<td>“The status field indicates the general health of a bug. Only certain status transitions are allowed.”</td>
</tr>
<tr>
<td>U</td>
<td>Unconfirmed</td>
<td>“This bug has recently been added to the database. Nobody has validated that this bug is true. Users who have the &quot;canconfirm&quot; permission set may confirm this bug, changing its state to NEW. Or, it may be directly resolved and marked RESOLVED.”</td>
</tr>
<tr>
<td>N</td>
<td>New</td>
<td>“This bug has recently been added to the assignee's list of bugs and must be processed. Bugs in this state may be accepted, and become ASSIGNED, passed on to someone else, and remain NEW, or resolved and marked RESOLVED.”</td>
</tr>
<tr>
<td>A</td>
<td>Assigned</td>
<td>“This bug is not yet resolved, but is assigned to the proper person. From here bugs can be given to another person and become NEW, or resolved and become RESOLVED.”</td>
</tr>
<tr>
<td>O</td>
<td>Reopened</td>
<td>“This bug was once resolved, but the resolution was deemed incorrect. For example, a WORKSFORME bug is REOPENED when more information shows up and the bug is now reproducible. From here bugs are either marked ASSIGNED or RESOLVED.”</td>
</tr>
<tr>
<td>R</td>
<td>Resolved</td>
<td>“A resolution has been taken, and it is awaiting verification by QA. From here bugs are either re-opened and become REOPENED, or are marked VERIFIED.”</td>
</tr>
</tbody>
</table>

\(^7\) Work on bug report state sequences was done with Les Gasser and Gabriel Ripoche. See Gasser & Ripoche (2003).
Table 4-2, cont.

<table>
<thead>
<tr>
<th>State Sequence Code</th>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Status:</strong></td>
<td>Code</td>
<td>Definition</td>
</tr>
<tr>
<td>V</td>
<td>Verified</td>
<td>“QA has looked at the bug and the resolution and agrees that the appropriate resolution has been taken. Any zombie bugs who choose to walk the earth again must do so by becoming REOPENED.”</td>
</tr>
<tr>
<td>C</td>
<td>Closed</td>
<td>This status value is in use, but is not listed or described in (Mozilla.org. A Bug's Life Cycle, 2003). Most bug reports are never moved from status=verified to status=closed.</td>
</tr>
</tbody>
</table>

**Resolution:** “The resolution field indicates what happened to this bug.”

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Fixed “A fix for this bug is checked into the tree and tested.”</td>
</tr>
<tr>
<td>I</td>
<td>Invalid “The problem described is not a bug”</td>
</tr>
<tr>
<td>X</td>
<td>Wontfix “The problem described is a bug which will never be fixed.”</td>
</tr>
<tr>
<td>D</td>
<td>Duplicate “The problem is a duplicate of an existing bug. Marking a bug duplicate requires the bug# of the duplicating bug and will at least put that bug number in the description field.”</td>
</tr>
<tr>
<td>W</td>
<td>Worksforme “All attempts at reproducing this bug were futile, reading the code produces no clues as to why this behavior would occur. If more information appears later, please re-assign the bug, for now, file it.”</td>
</tr>
<tr>
<td>M</td>
<td>Moved “The bug was specific to a particular Mozilla-based distribution and didn't affect mozilla.org code. The bug was moved to the bug database of the distributor of the affected Mozilla derivative.” (Mozilla.org. bugs, 2003.)</td>
</tr>
<tr>
<td>L</td>
<td>Later Deprecated. (Mozilla.org. bugs, 2003.)</td>
</tr>
<tr>
<td>R</td>
<td>Remind Deprecated. (Mozilla.org. bugs, 2003.)</td>
</tr>
</tbody>
</table>

**Other:** State Sequence Codes added in this project.

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>Undefined Represents an empty value in either the status or resolution field. Each state sequence begins “ZZ” indicating that when a bug report is created, it moves from an undefined status / resolution pair to some other valid status / resolution pair.</td>
</tr>
</tbody>
</table>

In Bugzilla, the status value represents “indicates the general health of a bug. Only certain status transitions are allowed” (Mozilla.org. A Bug’s Life Cycle, 2003). The resolution value “indicates what happened to this bug” (Mozilla.org. A Bug’s Life Cycle, 2003). The following table provides a summary of the state sequence codes used in this project, their mapping to the Bugzilla values, and their definitions in Bugzilla.

Unless otherwise noted, all quotes in the table above are from (Mozilla.org. A Bug’s Life Cycle, 2003).

Each state transition event is coded as a four-letter code (a quadruplet) using the pattern [initial_status] [initial_resolution] [revised_status] [revised_resolution]. Bug report
state sequences always begin in an undefined state: “ZZxx” where xx represents the initial settings of the status and resolution fields. The following example provides a text description of the state sequence that occurs most frequently in this sample:

**ZZNZ-NZRD-RDVD**

**Quadruplet 1:** Bug report begins Status=Undefined and Resolution=Undefined; it then moves to status=New and Resolution=Undefined.

**Quadruplet 2:** Bug report moves from status=New and Resolution=Undefined to status=Resolved and resolution=Duplicate.

**Quadruplet 3:** Bug report moves from status=Resolved and resolution=Duplicate to status=Verified and resolution=Duplicate.

Initial heuristics for determining whether a state sequence is normative or deviant are:

1. If the sequence occurs frequently, it’s “normal”
2. If the sequence moves forward and results in closure, it’s “normal”
3. If the sequence includes re-opening, it’s “deviant”
4. If the sequence includes retrograde movement, and is not re-opened it’s “sub-optimal”
5. If the sequence includes both re-opening AND retrograde movement, it’s “deviant”

The state sequences are also evaluated for completeness, where this is a measure of whether the BR has been set into a “terminal” status-resolution pair. Bug reports whose final status value is “verified” or “closed” are considered “completed.” Bug reports whose final status value is “resolved” are considered “nearly completed.” Bug reports
whose last status value is “unconfirmed,” “new,” “assigned” or “reopened” are considered “incomplete.”

Table 4-3 Bug Report Completion States

<table>
<thead>
<tr>
<th>Completed</th>
<th>Nearly Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>VD - verified duplicate</td>
<td>RD - resolved duplicate</td>
</tr>
<tr>
<td>CD - closed duplicate</td>
<td>RF - resolved fixed</td>
</tr>
<tr>
<td>VF - verified fixed</td>
<td>RI - resolved invalid</td>
</tr>
<tr>
<td>CF - closed fixed</td>
<td>RW - resolved works for me</td>
</tr>
<tr>
<td>VI - verified invalid</td>
<td>RX - resolved won’t fix</td>
</tr>
</tbody>
</table>

Table 4-4 Bug Report State Sequences: Patterns and Frequencies

<table>
<thead>
<tr>
<th>State Sequence Pattern</th>
<th>Frequency</th>
<th>Normativity</th>
<th>Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZZNZ-NZRD-RDVD</td>
<td>40</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ-UZRD-RDVD</td>
<td>33</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZAZ-AZRF-RFVF</td>
<td>26</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ</td>
<td>17</td>
<td>Normative</td>
<td>Incomplete</td>
</tr>
<tr>
<td>ZZNZ-NZRF-RFVF</td>
<td>17</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ-UZRD</td>
<td>16</td>
<td>Normative</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ-UZNZ</td>
<td>15</td>
<td>Normative</td>
<td>Incomplete</td>
</tr>
<tr>
<td>ZZUZ-UZRW</td>
<td>14</td>
<td>Normative</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ-UZRW-RWVW</td>
<td>12</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ</td>
<td>10</td>
<td>Normative</td>
<td>Incomplete</td>
</tr>
<tr>
<td>ZZUZ-UZRI</td>
<td>10</td>
<td>Normative</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ-UZNZ-NZRD-RDVD</td>
<td>10</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ</td>
<td>7</td>
<td>Normative</td>
<td>Incomplete</td>
</tr>
<tr>
<td>ZZUZ-UZNZ-UZRI</td>
<td>7</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ-UZNZ-NZRF-RFVF</td>
<td>7</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ</td>
<td>6</td>
<td>Normative</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZNZ-NZAZ-AZRF</td>
<td>6</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ-UZRF-RFVF</td>
<td>6</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ</td>
<td>5</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZRI-RVI</td>
<td>5</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZAZ-AZRD-RDVD</td>
<td>3</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ</td>
<td>3</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZRF-RWVV</td>
<td>4</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ</td>
<td>4</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ-UZNZ-NZAZ-AZRF</td>
<td>4</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ</td>
<td>4</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ-UZNZ-NZAZ-AZRF-RFVF</td>
<td>4</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ</td>
<td>3</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ</td>
<td>3</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ</td>
<td>3</td>
<td>Normative</td>
<td>Completed</td>
</tr>
</tbody>
</table>

123
<table>
<thead>
<tr>
<th>State Sequence Pattern</th>
<th>Frequency</th>
<th>Normativity</th>
<th>Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Incomplete</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Normative</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ.UZNZ.NZRF-RRF</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
</tbody>
</table>

Table 4-4, cont.
<table>
<thead>
<tr>
<th>State Sequence Pattern</th>
<th>Frequency</th>
<th>Normativity</th>
<th>Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZZUZ-UZNZ-NZRW-RWOZ-OZNZ-NZRD</td>
<td>1</td>
<td>Deviant</td>
<td>Nearly Completed</td>
</tr>
<tr>
<td>ZZUZ-UZNZ-NZRW-RWOZ-OZRD-RDVD</td>
<td>1</td>
<td>Deviant</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ-URZD-RDUZ-UZNZ-NZAZ-AZRF-RFVF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ-UZRW-RWUZ-UZNZ-NZAZ-AZRF-RFVF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZUZ-UZNZ-NZAZ-NZNZ-NZAZ-NZAZ</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Incomplete</td>
</tr>
<tr>
<td>ZZNZ-NZRI-RFOZ-OZNZ-NZAZ-AZRF-RFVF</td>
<td>1</td>
<td>Deviant</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZRW-RWOZ-OZNZ-NZAZ-AZRF-RFVF</td>
<td>1</td>
<td>Deviant</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZRF-RFOZ-RFOZ-OZRF-OZRF-RFVF</td>
<td>1</td>
<td>Deviant</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZAZ-AZRF-RFVF-VFOZ-OZRF-RFVF</td>
<td>1</td>
<td>Deviant</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZAZ-AZRF-RFOZ-OZRF-OZRF-OZRF-RFVF</td>
<td>1</td>
<td>Deviant</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZRW-RWOZ-OZNZ-NZRW-RWOZ-OZRW-RWVW</td>
<td>1</td>
<td>Deviant</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZAZ-AZNZ-NZAZ-AZNZ-AZNZ-NZAZ-AZRF-RFVF</td>
<td>1</td>
<td>Sub-optimal</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZAZ-AZNZ-NZAZ-AZRD-ODAD-ODAD-ADAZ-AZRW-RWVW</td>
<td>1</td>
<td>Deviant</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZAZ-AZRF-RFOZ-OZNZ-NZAZ-NZRF-RFOF-OZRF-RFVF</td>
<td>1</td>
<td>Deviant</td>
<td>Completed</td>
</tr>
<tr>
<td>ZZNZ-NZRF-RFOZ-OFNZ-AZRF-AZRF-RFVF</td>
<td>1</td>
<td>Deviant</td>
<td>Completed</td>
</tr>
</tbody>
</table>

* Null: this is a bug report describing a security problem. The bug report was unavailable for analysis.
** Unknown: these bug reports were apparently corrupted and data was lost; therefore the state sequence data is unreliable.

**Dialogs**

A dialog, in the context of this project, is an exchange of ideas, information, or opinion, expressed as statements, within the comments appended to a bug report. The community members often enter comment text that (1) refers or responds to statements made earlier or (2) is intended to elicit a subsequent response from other community members. Two forms of dialog have been identified in the bug reports: question and answers and negotiation, both of which are described below.
Question and Answer

Question and answer (Q&A) dialogs occur frequently in the bug report sample drawn from the F/OSS bug report repository studied in this project.

There are two distinct patterns of Q&A in the bug reports. The first pattern is characterized by a pair (or set?) of statements that are concerned with the initial query posed by the questioner and responses submitted by respondents.

Table 4-5 Question and Answer Patterns

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question posed in one statement</td>
<td>No answer</td>
</tr>
<tr>
<td>Question posed in one statement</td>
<td>Answer given in one statement</td>
</tr>
<tr>
<td>Question posed in one statement</td>
<td>Answer given in &gt; 1 statement, possibly by &gt; 1 respondent</td>
</tr>
<tr>
<td>Question posed in &gt; 1 statement</td>
<td>No answer</td>
</tr>
<tr>
<td>Question posed in &gt; 1 statement</td>
<td>Answer given in one statement</td>
</tr>
<tr>
<td>Question posed in &gt; 1 statement</td>
<td>Answer given in &gt; 1 statement, possibly by &gt; 1 respondent</td>
</tr>
<tr>
<td>No explicit question posed</td>
<td>Answer given to implicit question</td>
</tr>
</tbody>
</table>

Note that in the current study, instances of Q&A were only examined within the bounds of individual bug reports. When questions appeared within BR networks, no attempt to locate response information within the other bug reports within the BRNs was made.

Negotiation

Negotiation is a core concept that has emerged from this analysis. Strauss (1978) defines negotiation as a process:

... when individuals or groups or organizations of any size work together 'to get things done' then agreement is required about such matters as what, how, when, where, and how much. Continued agreement itself may be something to be worked
at…. Putting the matter thus suggests that negotiations pertain to the ordering and articulation of an enormous variety of activities (p. ix).

Negotiation is a basic social process (Strauss, 1978) and its manifestation in this community’s bug reports is discussed in greater detail in the Chapter Seven.

Conclusion

This chapter identified a set of potential units of analysis related to the information practices employed by this F/OSS development community as it performs its SWPM work. These units of analysis are society at large, F/OSS meta communities, F/OSS communities, bug report repositories, bug report networks (BRNs), bug reports, bug report state sequences, dialogs (such as question and answer sequences and instances of negotiation), comments, and statements. Some of these forms of information, activities, and concepts were treated in this chapter and will not be addressed in any more detail in this report (society at large, F/OSS meta communities, F/OSS communities beyond the one whose bug reports are analyzed here, and bug report state sequences, comments, and statements). The other units of analysis identified here are treated in the remaining chapters. Chapter Five focuses on bug reports, the first-class information object that is the fundamental unit of analysis in this study, and the virtual information object, bug report networks, created as community members assert relationships between bug reports. Chapter Six examines the influence of context upon SWPM work and the information practices that arise within those contexts. Chapter Seven examines the interrelated, nested processes, including triage and negotiation, employed in SWPM. Chapter Eight examines relationships between information practices and the community’s social order.
CHAPTER FIVE: BUGS, BUG REPORTS, AND BUG REPORT NETWORKS

Introduction

This chapter clarifies the critical distinction between a bug, a phenomenological event, and bug reports, an information object that is used to represent those phenomena. The chapter then turns to an analysis of bug report networks (BRNs), which are collections of bug reports created by community members to support software problem management (SWPM). BRNs are created as community members asset formal and informal relationships between bug reports. Linking bug reports together is identified as one of the fundamental information practices employed by this community to support their SWPM process.

Bug Reports Are Not Bugs

A critical distinction in the examination of SWPM is that between a software problem, or bug, and a bug report. One definition of a bug, or software problem, is “an unexpected defect, fault, flaw, or imperfection” (MerriamWebster.com, 2004). Kajko-Mattsson (1999) defines a software problem as “a human encounter with software causing a difficulty, doubt, or uncertainty” (p.169). Thus, for the purposes of this project, a bug (or software problem) is defined as a phenomenological situation, generally lacking tangible form, in which software system behavior fails to conform to some person’s or group’s expectations and values in a manner judged to have negative sentimental, moral, or practical import.

A bug report is a first-class information object in this community’s bug report repository. Bug reports are created after a problematic human-system encounter occurs and the human encountering the problematic phenomena is motivated to create a bug report. (Bugs may be encountered and not reported: if they are not reported, there is little chance that the bug will ever be corrected.) A bug report is created in order to provide a
formal and tangible representation of the bug to support the process of correcting the problem which it represents.

Bug reports are the central artifact supporting SWPM within this F/OSS development community. Bug reports represent many different things (details about the structure and content of the bug reports used by this F/OSS development community are provided in Chapter Four). A bug report contains descriptive information about the bug. It also contains instructions for how other people can make the bug happen so it can be recreated because bug (phenomena) reproducibility is a desirable characteristic. Bug reports contain information about what work has been done, what work is underway, and what work is planned or committed to in the future. A bug report holds information about who has been involved in the reporting, triage, analysis, coding, implementation, and verification of the bug and its associated fix. Bug reports timestamp changes and entries made to the bug report and identify the person who made the change, supporting accountability of action. Bug reports often also contain opinion, hypotheses, clarifications, and conjecture, entered as text and made available to anyone in the community.

The bug report supports re-creation of the problem situation – the bug – so that the person(s) or organization(s) responsible for fixing the bug have sufficient information to develop and test a resolution. The bug report makes the problem visible to other people and organizations and supports a coordinated chain of activity that should result in the fixing of the associated software problem. The bug report is a nexus of information and activity while the bug is fixed. The bug report carries a series of annotations that describe past events and point to future work.

In different situations, members of the community may be working on either a bug or on a bug report. If a programmer is making and testing code changes, he is working on a bug. If a manager is evaluating whether a new bug report is a duplicate of another existing bug report, she is reading, evaluating, managing, and linking bug reports in the repository. The distinction between bug reports and bugs is critical to our analysis because typical collective repository tools support bug reports as first class objects, but
the aims of analysis and action are bugs, which are not first-class objects in repository tools.

The normative expectation is that there is a one-to-one relationship between bugs and bug reports. The evidence from the examination of the bug report sample reported on here shows a high degree of variation. The expected one-to-one relationship is common, but others occur frequently as well. For example, many bug reports may contribute to the understanding or resolution of a single bug (Gasser & Ripoche, 2003). The relationships between bugs and bug reports seen in the data include:

- One bug to one bug report: the normative relationship
- Many bugs to one bug report
  - variation 1: BRN representing a dependency chain, where resolution of the dependent problem must wait until resolution of the pre-requisite problem is complete
  - variation 2: BRN organized around a meta bug report
  - variation 3: a complex problem phenomenon described in a single bug report; complex bug reports are often re-factored into two or more bug reports
- One bug to many bug reports: a BRN of duplicates
- No bug to one bug report: the phenomenon described by the reporter in the bug report is determined to not represent a problem to be solved (such bug reports are often marked resolved with resolutions like invalid, worksforme, or wontfix)

**Bug Report Networks**

The empirical research performed as part of this project has shown that a notable structural feature of defect tracking repositories is the evolving "bug report network" (BRN). Community members create BRNs by progressively asserting various formal and informal relationships between bug reports (BRs). In this F/OSS bug report repository,
community members assert two types of formal, symmetrical relationships (*duplications* and *dependencies*) and various *informal relationships* (like "see also" references) between bug reports. About two-thirds (65%) of the bug reports in this repository are associated with other bug reports using one of these three types of relationships. BRNs can be interpreted as (1) information ordering strategies that support collocation of related bug reports, decreasing cognitive and organizational effort of the community members; (2) sense-making strategies wherein BRNs provide more refined representations of software and work-organization issues; (3) social ordering strategies that rearrange collective relationships among community members. This section presents findings about the nature, extent, and impact of BRNs in this F/OSS development community.

Figure 5-1 Bug Report Repository Patterns

Bug reports are first-class database objects, but bug report networks are not. Figure 5-1 shows three typical BRN patterns. The BRN on the left represents the 35% of the bug reports that have zero formal or informal relationships with other bug reports: each of these bug reports forms a trivial BRN. The middle BRN shows two bug reports associated by a dependency or informal relationship. The BRN on the right shows a more
complex set of relationships. The doubled bug report (the rightmost bug report) represents a bug report with a duplicate relationship to the second bug report behind it. The duplicated bug report is also associated by either a dependency or informal relationship with the other bug report to its left in the same BRN. Note that it is also possible for BRNs to be connected to each other, as indicated by the line connecting the middle BRN to the rightmost BRN.

We use the following definitions, based upon the definitions stated by the community in their software problem management documentation, to identify the formal relationships between bug reports:

- **Duplicate**: A bug report is marked as a duplicate if the problem represented by the bug report is believed to be already represented by another existing bug report. A duplicate relationship is a formal, symmetrical relationship between two bug reports.

- **Dependency**: A bug report is marked as a blocker of another bug report if resolution of the software problem it represents blocks development and/or testing work on the problem represented by the other bug report. A bug report is marked as dependent on another bug report if the problem it represents can't be fixed until the problem represented by the other bug report is fixed. Bug reports that are dependent on each other have a formal, symmetrical “blocks” / “depends on” relationship.

Besides the two formal relationships described above, community members frequently create informal relationships, like “see also” references, by referring to other bug reports when they are adding text comments to existing bug reports. While it is possible to automatically extract instances of informal relationships from the repository, the nature and purpose of these references vary considerably and, at this point, are most reliably understood by reading the bug reports and understanding the contexts in which the citations are made. Here are some examples of these informal references:
- This looks related to #X (Bug report 9)
- See comments on X -- same applies here I think. (Bug report 21)
- My fix for X kinda helps fixing this too. (Bug report 21)
- Could be dup of bug X or bug Y. (Bug report 35)
- Should bug X be added to this? (Bug report 220)
- in bug X comment 1, I said this works in 4xp. (Bug report 265)

Figure 5-2 represents the bug report network associated with one “critical” severity bug report (BR-B) drawn from the bug report repository under study. This bug report network, consisting of six bug reports, illustrates a number of different relationships that often occur in this repository. The x-axis represents time over a 6-month period; the relationship of the objects in the diagram to the timescale is approximate. The columns of dashes below each bug report’s lifeline represent the count of comments added to a bug report on a single day and are shown to provide a sense of the level of activity associated with each bug report throughout the bug report’s life.

Bug report “B” (BR-B in Figure 5-2) is the central report in this network. BR-B was opened with a “critical” severity level because it represented a bug that caused the software system to crash. As soon as it was opened, it was associated as “blocking” the resolution of BR-A. BR-A already existed, and was defined as a meta bug report used to collocate a group of 15 bug reports representing bugs that caused crashes in this part of the overall system. (Note that it would be possible to look at BR-A as the central bug report in a different BRN: this is an example of how BRNs can be connected to each other as shown in Figure 5-1.) Meta bug reports are an informal adaptation of the bug report repository made by the community in order to increase the utility of the bug report repository. The community defines a meta bug report as “A placeholder bug [report] for tracking the progress of other bug [report]s. Meta bug [report]s are made dependent on other bug [report]s so that interested parties can be kept up-to-date with status via one bug [report], without having to receive all the mails related to all the bug [report]s related
to the development of a particular area” (Mozilla.org, Bugzilla Keyword Descriptions List). By convention, users mark bug reports of this type by entering the string “meta” in the keyword field of the bug report.

The chain of duplicate relations between BR-B, BR-C, and BR-D is of interest. BR-C was opened several weeks after BR-B, during a six-week period when BR-B was not very active (no comments added). BR-D was opened later the same day BR-C was opened. BR-D was quickly identified as representing the same bug as BR-C and marked “resolved/duplicate.” BR-C was not identified as representing the same phenomena as BR-B until about six weeks after BR-C was opened.

Figure 5-2 Bug Report Network
The relationships between BR-B, BR-E, and BF-F are also of interest. The level of activity on BR-B was high during month 5. At one point, a patch for the bug represented by BR-B was introduced. This change caused the bug represented by BR-E (a type of bug and bug report identified as a “regression”) to occur. BR-F was opened a couple of hours after BR-E was opened, and was immediately recognized as a duplicate of BR-E. The bad patch associated with BR-B was quickly backed out to resolve the problem associated with BR-E. BR-E was then marked “resolved/fixed.” About one month later, the problem represented by BR-B was finally resolved.

**Varieties of Relationships between Bug Reports**

The various kinds of relationships that occur between bug reports and their frequency of occurrence are described in this section. About two-thirds (65%) of the 385 bug reports in the sample have either a formal or informal relationship with at least one other bug report. Table 5-1 shows the frequency with which different types of relationships occur within the sample of 385 bug reports. Note that a single bug report can have multiple types of dependencies: a bug report can have informal relationships with other bug reports and simultaneously have dependency and duplicate relationships.

The community has established conventions regarding bug report networks. Community members sometimes create bug reports that, instead of representing problems (bugs), represent a collection of bug reports having common characteristics (e.g., all the high priority bug reports that should be fixed prior to the next software release). Community members refer to these bug reports as “meta” or “tracking” bug reports. In the BRN illustrated in Figure 5-2, BR-A is a “meta” bug report used to create a collection of bug reports having a common characteristic: bug reports representing system crashes of a similar type. BRNs centered around meta bug reports, like all bug report networks, are not first-class database objects. They represent a specific social and information management adaptation made by community members to increase the utility of the bug report repository. As is typically done, the meta bug report has 15 different
bug reports “block” the “resolution” of this bug report. The meta bug report, of course, doesn’t represent a bug that can be repaired in any normal sense of the word.

Table 5-1 Frequency of BRN relationships

<table>
<thead>
<tr>
<th>Duplicates</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BRs with one or more duplicate BRs</td>
<td>10%</td>
</tr>
<tr>
<td>BRs resolved as a duplicate of another BR</td>
<td>33%</td>
</tr>
<tr>
<td>Dependencies</td>
<td></td>
</tr>
<tr>
<td>BRs “blocking” one or more BR</td>
<td>12%</td>
</tr>
<tr>
<td>BRs “dependent on” one or more BR</td>
<td>7%</td>
</tr>
<tr>
<td>Informal</td>
<td></td>
</tr>
<tr>
<td>BRs with “informal” relation to one or more BR</td>
<td>33%</td>
</tr>
</tbody>
</table>

Impacts of Bug Report Networks on the Software Problem Management Process

The creation of BRNs is one common way the community responds to the information management challenges presented as they execute their SWPM process (see Table 5-1). This section discusses what motivates community members to assert these relationships between bug reports. BRNs can be interpreted as (1) information management strategies that support collocation of related bug reports, decreasing cognitive and organizational effort of the community members; (2) sense-making strategies wherein BRNs provide more refined representations of software and work-organization issues; (3) social ordering strategies that rearrange collective relationships among community members.

The creation and evolution of BRNs represents an information ordering strategy as bug reports are collocated as duplicates or dependents. Accurate identification of duplicate bug reports is a critical method by which the community reduces the number of bug reports requiring attention by software developers, a scarce and valuable resource. Based upon typical rates of bug report creation (on the order of 200 per day) and the frequency with which bug reports are marked duplicates of existing bug reports (33% of
all bug reports in the sample are eventually marked as a duplicate of another bug report), this strategy can reduce the number of bug reports requiring additional investigation from about 200 to approximately 135 per day.

Shrinking the set of bug reports to work on reduces the size and complexity of the field of work. However, identification of duplicates implies a productivity cost because duplicates must be identified by members of the community. There is also danger of mis-identification: bug reports that are not true duplicates (false positives) will be ignored because their status is “resolved.” It’s also clear, because of late-marked duplication, that “undiscovered” duplicate bug reports exist and multiple groups of people may be duplicating effort by working on two bug reports that represent the same issue (see, for example, the time period between the opening of BR-C and its resolution as a duplicate of BR-B in Figure 5-2).

When a bug report is marked "resolved/duplicate" this means the bug report is resolved but it does not mean that the underlying bug itself has been resolved. "Resolved/duplicate" means that the resolver(s) believe this is a duplicate report of a phenomenon that already has an effective representation elsewhere in the repository. It doesn't even mean that that the resolved bug report can now be ignored, since we have seen instances of late-identification of duplicates (e.g., BR-C in Figure 5-2) in which accumulated knowledge and dialogue may still be relevant to the resolution of the other bug reports in the BRN. Thus the semantics of the “resolved” keyword are more complex than they might first appear.

The establishment of dependency relationships configures future work in a particular way: problem resolution activities, often by different developers, are configured in a sequence to ensure that the pre-requisite problems are resolved first. Meta bug reports are a special case of the application of dependency relations: this practice is a strategy for reducing the cognitive effort of individuals who wish to monitor progress on a set of bug reports that, while not sequentially dependent upon each other, have some other factors in common that are of importance to the community. Dependency relationships, whether a meta bug report relationship or not, arrange bug reports in ways
that provide a more accurate mapping of the bug reports to the requirements of solving problems and the community’s conventions regarding the of organization of work.

Establishment of dependency relationships between bug reports is one way SWPM is revealed as a sensemaking enterprise (Weick, 1995). Sensemaking is a process used by organizations to structure the unknown and helps explain how people generate and structure information so they can interpret situations of uncertainty or misalignment. Constructing BRNs is an information structuring strategy by which individual bug reports, first-class database objects, are composed into a new form of information, the BRN. Creating a BRN collocates a group of bug reports that would otherwise remain scattered and disassociated from each other, adding virtual structure to the bug report repository. The informal, “see also” relationships that are asserted between bug reports illustrate one way in which community members express their personal knowledge of the software system as a whole, their cognate areas of expertise, and their knowledge of the content of other existing bug reports.

Establishment and manipulation of BRNs can be interpreted through the lens of Weick’s seven properties of sensemaking (1995), three of which are discussed here. First, BRNs are enactive of sensible environments. Community members understand that creating a BRN isn’t going to produce a change in the code that resolves a software problem. They are making moves to organize the information available, using the tools at hand, in a way that supports the community’s collective effort toward solving the underlying problem (or at least moving a bug report into the set of bug reports no longer requiring attention, as occurs when a bug report is marked as a duplicate). Creating a dependency relationship is an activity of relating information objects together, organizing the relevant information objects into relationships that retrospectively mirror the community’s perception of the problematic situation and guide its future activities. Second, creation of BRNs, like sensemaking endeavors generally, is social. Informal relationships (“See comments on X – same applies here I think” or “My fix for X kinda helps fixing this too” (from Bug report 21)) are signals to other readers in the community to consider the stories and trajectories expressed in other bug reports. The composer of such comments isn’t making a claim of
a firm, formal dependency or duplicate relationship, but seems to be relating other experiences that may provide insights to the current problem and make resolution easier and faster to achieve. Also, once a community member is an active participant in the SWPM process, no one is particularly privileged by their role in the community to assert relationships between bug reports. That is, you don’t have to be the assignee or QAContact to make these kinds of assertions. Third, creation of BRNs is provisional, driven by plausibility rather than accuracy. A repository of hundreds of thousands of bug reports is a landscape filled with items of roughly the same shape. BRNs, by collocating specific bug reports, act as a filter to help community members avoid being overwhelmed by undifferentiated data. Assertion of relationships between bug reports is also a way for community members to take action before complete information on how to code a solution to a problem is available: it is part of enacting the actions and information required to solve the problem. New information may contravene older information: it is not uncommon for early assertions of dependency or duplication to be retracted in the face of continuing investigation into a software problem. “A good story holds disparate elements together long enough to energize and guide action, plausibly enough to allow people to make retrospective sense of whatever happens, and engagingly enough that others will contribute their own inputs in the interest of sensemaking” (Weick, 1995) (p.61).

Bug reports are more than just information or a locus of information management. They are also specifications/codifications of social relationships, such as roles (reporter, assigned-to, cc: list member) and dynamic and patterned interactions (e.g. dialogues, question-response or question-response-elaboration sequences; other basic social processes such as negotiation and articulation of work, etc.). This means that as information is ordered through BRN creation/extension/modification, social relations are also being ordered (organized, established, patterned). Consider the potential impact of asserting a dependency relationship between two bug reports that are assigned to two different programmers. In this situation, two programmers, plus the other people working on each of the bug reports, have been suddenly brought into closer association
with each other. The programmer whose work blocks the other programmer’s work experiences additional pressure to complete the fix. There may also be a negative impact on time to resolution because more people are involved, more articulation work may be required, and it may be more likely that situations arise that must be negotiated among a larger group of people.

Conclusion

This chapter clarified the critical distinction between a bug, a phenomenological event, and bug reports, an information object that is used to represent those phenomena. It then analyzed bug report networks (BRNs), collections of bug reports created by community members to support software problem management (SWPM). BRNs are frequently created as community members assert formal and informal relationships between bug reports. Linking bug reports together is identified as one of the fundamental information practices employed by this community to support their SWPM process. About two-thirds (65%) of the bug reports contain one or more formal or informal references to other bug reports. BRNs represent (1) information ordering strategies that support collocation of related bug reports, decreasing cognitive and organizational effort of the community members; (2) sense-making strategies wherein BRNs provide more refined representations of software and work-organization issues; and (3) social ordering strategies that rearrange collective relationships among community members.

A number of practical and research issues remain to be addressed in regard to the way the bug report repository supports the community’s information management and problem resolution activities. In terms of the practical issues, the bug report repository provides limited support for the management of BRNs as information objects. First, duplicates and dependencies must be identified and marked manually with only the support of the repository’s search interface. Automatic processes for the identification of potential duplicate and dependent bug reports would make the community’s SWPM process more efficient. Second, the current repository implementation provides simple
visualization support for only dependency relationships: the common and important
duplicate and informal relationships are not represented at all in the visualizations. The
element of time and an indication of the relative levels of activity, as shown in Figure 5-2,
are not available either. Improved visualization tools could provide easier-to-understand
summaries of the often complex relationships between bug reports. Third, while bug
reports are first class information objects and therefore have associated direct operations
(like setting the bug report to a resolved status) and data elements, BRNs cannot be
managed or tracked in analogous ways. It is not possible to mark a BRN as “resolved,”
nor is it possible to search for BRNs in any direct way (meta bug reports anchoring
dependency BRNs are the only known exception but only if the convention or using the
“meta” keyword is observed). Support for BRNs as first class objects with related
behavior can widen the community’s repertoire of techniques for managing software
problems.

Several important research questions still remain to be addressed. First, to
compensate for the limitations of the research method employed here, other techniques,
such as interviewing or participant/observer methods, should be employed to understand
the situations in which the information represented by BRNs is helpful or unhelpful in
managing software problems. We would also like to understand the extent to which
complex BRNs are taken into account by community members during problem
resolution. Second, comparative studies should also be performed to determine whether
the information management practices identified here are employed by other F/OSS
communities or in traditional, closed software development communities. Third, there
may be patterns of BRNs, for example patterns in the kinds of links that appear, and
patterns in the types of links that are sanctioned and even crystallized into standard
categories and supporting tools, such as dependencies and duplicates. There may also be
patterns in how such networks are formed and how they evolve. Finally, we would like to
determine how the inclusion of a BR in a BRN affects the community’s SWPM
performance (e.g., testing for correlation between the membership of a bug report in a
BRN and time to resolution for individual bug reports).
Linking of information objects is a candidate for inclusion in the set of fundamental human and collective information practices. Other F/OSS and traditional software development communities should be examined in order to determine if it is a common phenomenon in software problem management settings. Investigation of other communities engaged in SWPM (and investigation of communities involved in other kinds of activities, like medical diagnosis, digital library management, or real time infrastructure management) will also help us determine if linking of information objects warrants description as a fundamental human information practices.
CHAPTER SIX: LAYERS OF CONTEXT IN DISTRIBUTED WORK

Introduction

This chapter discusses how context influences the organization and enactment of distributed work in this community. Here, context means the set of structural (e.g., technical, economic, organizational and political), meso-level (e.g., bug report repository and bug report networks), and micro-level (e.g., individual bug reports and instances of negotiation) constraints, boundaries and assumptions that influence the nature of distributed work in this F/OSS development community.

An example of one broad structural context is the F/OSS development model upon which the community studied here is based. Examples of meso-level contexts are the coordination mechanism (the bug report repository and the community’s software problem management (SWPM) protocol), within which this community’s SWPM work is conducted. Micro-level contexts include individual bug report networks (BRNs), individual bug reports and instances of negotiation associated with individual bug reports. Structural, meso-level and micro-level contexts exist and interact simultaneously, creating a set of overlapping layers of context. All of these layers of context are important because they constrain or limit activity, process, and social order in the distributed work community. If these constraints were different, it is likely that the nature of the distributed work within the community, including the community’s information practices, would also be noticeably different.

This chapter argues that the process of SWPM in this community is in part a function of the layers of context that hold at a given time and in the context of particular bug reports. The findings presented in this chapter, particularly those related to the meso- and micro-level contexts, are grounded in the data analyzed in this project. The structural level contexts, on the other hand, were not a direct focus of this project and are more tentatively grounded in this project’s data. Identification of these contexts and their associated constraints is valuable because it simultaneously signals the limitations of this
particular study and suggests how future research projects might be designed in order to provide more information on the significance of each of the contexts identified here. The current project is a case study of a single F/OSS development community, so the assertions regarding the contexts identified here are provisional. Further research is needed to fully understand the validity and importance of this set of constraints across multiple settings.

Particularly significant elements of context are discussed in this chapter. The structural contexts (technical, economic, organizational, and political/ideological) identified in this project are discussed first, followed by a discussion of one particular meso-level context (the community’s bug report repository) and finally a review of selected micro-level contexts (bug report networks, bug reports and instances of negotiation).

**Structural Contexts**

Strauss, in his book on negotiation, uses the term *structural context* to describe “that ‘within which’ the negotiations take place, in the largest sense” (1978) (p. 98). This project takes a wider view, considering negotiation as one of several basic processes that are a part of SWPM. Thus, if we replace the word “negotiation” with “distributed work” in the preceding quote, we can adapt Strauss’ notion of structural context to the examination of this software development community. By contrast, later sections of this chapter discuss meso- and micro-level contexts, within which distributed work takes place in smaller and the smallest senses.

In Strauss’ view (1978), social order is negotiated order, and social order is negotiated within a set of contexts. Negotiated order refers to the situation that holds whenever people use negotiation, one process by which people work together to “get things done, to reach common as well as private goals” (p.121). Day and Day (1977) (as quoted in Strauss, 1978, p.260), provide this summary of negotiated order:
In the case of negotiated order theory, the individuals in organizations play an active, self-conscious role in the shaping of the social order. Their day-to-day interactions, agreements, temporary refusals, and changing definitions of the situations at hand are of paramount importance. Closely correlated is the perspective’s view of social reality … the negotiated order theory downplays the notions of organizations as fixed, rather rigid systems which are highly constrained by strict rules, regulations, goals and hierarchical chains of command. Instead, it emphasizes the fluid, continuously emerging qualities of the organization, the changing web of interactions woven among its members, and it suggests that order is something at which the members of the organization must constantly work. Consequently, conflict and change are just as much a part of organizational life as consensus and stability. Organizations are thus viewed as complex and highly fragile social constructions of reality which are subject to the numerous temporal, spatial, and situational events occurring both internally and externally. (Day and Day (1977) p.132; cited in Strauss (1978), p. 260)

In the case of this F/OSS development community and its SWPM activities, elements of the structural context include technical context, economic context, organizational context, and political / ideological context.

**Technical Context**

The technical context in this community includes the specific tools and infrastructure used to support SWPM. Some of these include:

1. The other development tools, including the bug report repository (Bugzilla), the code repository (CVS, the concurrent versioning system); test tools; computer languages and compilers
2. The communications infrastructure (Internet; World Wide Web)
3. Software standards defined by various organizations

The technical apparatus of the bug report repository provides (and, conversely, doesn’t provide) particular affordances to the development community. The bug report repository does not contain software problems. The repository instead holds bug reports,
which are *representations of software problems*, and thus much of the work of SWPM is the management of bug reports, and only indirectly the management of software problems.\(^8\)

The bug report repository used by the community studied here (Bugzilla) is designed to create and distribute e-mail to the community members who have registered an interest in a particular bug report, or class of bug reports, when a bug report they are registered with changes. This is a particularly valuable affordance in a community that is large, globally distributed, and must contend with a high bug report creation rate.

The information and communications technology infrastructure available to support the community also plays a role. The ubiquitous availability of the Internet, e-mail, and the World Wide Web enable the effective deployment of this community’s bug report repository in its current form. It is easy to imagine how the infrastructure technology, if somewhat different, would affect the SWPM practices of the community. Consider, for example, how different the exchange of screen shots would be without the availability of graphical World Wide Web browsers. Or consider how notifications about software problems and bug reports would be transmitted to community members if a single, unified e-mail routing scheme were not available. See Carstensen, Sørensen & Tuikka (1995) for a description of a paper-based bug report repository and SWPM process.

The other particular development tools in the community are also important aspects of technological context (e.g., computer languages, code repositories, test tools, etc.) but are beyond the scope of this project. However, selection of different tools and languages might change the membership of the community in unpredictable ways. Use of uncommon tools and languages in a project would likely inhibit participation by some people who might otherwise be interested in participating in the project. Conversely, selection of commonly used tools and languages makes participation easier for a larger group of participants by lowering access barriers and flattening learning curves.

---

8 It is difficult to imagine how software problems could be represented directly in a bug report repository. The direct representation of a software problem is in the code-in-use.
Finally, the availability of and commitment by this community to particular computing standards also shapes the technical context. In many cases, getting the system’s behavior into compliance with established published standards drives the creation of particular bug reports regarding the system’s non-compliance with standards embraced by the community. Adoption of a different set of standards or a different philosophical outlook toward standards compliance would likely affect the system’s design, the kinds of software problems reported and the rate of system adoption by the user community.

Economic Context

Detailed consideration of the economic contexts is beyond the scope of this project. There is, however, some relationship between the economic context and distributed SWPM. The economic contexts affecting this community include:

1. The conditions of the organization of labor.
2. The nature of the F/OSS “marketplace” and its products

The conditions of the organization of labor for this project are complex. In addition to a very large community of users and a large community of volunteer developers, there is a core group of community members who are or have been employed by the community itself or its commercial sponsors and affiliates. This mixed mode of labor organization varies from the popular but overly simplistic view of F/OSS projects as comprised only of volunteers.

The nature of the F/OSS marketplace also influences SWPM in several ways. The fact that users can download and use the software systems without paying license fees has helped build the user base to a size that is far greater than the size of the development community. The openness of the community, as represented by global
access to the code and bug report repositories, enables any of the systems’ users to submit bug reports if they are willing to register as users of this community’s instance of Bugzilla.

The software products produced by this community are available for use by anyone at no cost. The system is also free in the sense that, because the source code is available, anyone with the inclination and motivation is free to modify the system itself, either contributing to the main development branch, or to create a more-or-less new system based upon the software already produced by the community (the creation of a “new” system is sometimes called “forking,” and is not always viewed as a constructive turn of events). In some F/OSS projects, systems are produced in a modular fashion to allow users options in selecting and composing systems from the modules available. The Mozilla project, for example, produces a browser integrated with e-mail and chat clients and an HTML editor (the Mozilla browser suite). The Mozilla project also produces a stand-alone browser (Firefox) and a stand-alone e-mail client (Thunderbird) which are essentially the browser and e-mail client components of the integrated browser suite reincarnated as stand-alone applications. The Mozilla browser suite is also designed to be a core browser / HTML rendering system that can be re-packaged as another product. Netscape has released several versions of the Netscape browser that are, at their core, Mozilla.

Organizational Context

The organizational context in this community includes:

1. The particular way in which the F/OSS community is organized (democratic / flat / hierarchy / module owners / hybrid with commercial firms).
2. The nature and influence of “meritocracy” in the community.
3. The nature and influence of “bureaucracy” in the community.
4. The nature and influence of software engineering “best practices” in the community.

F/OSS software projects employ a variety of organizational forms, and detailed discussion of these varieties is beyond the scope of this project. In general, however, the labor of production (design, coding, fixing problems) is largely donated by volunteers, which constrains who can and will participate, and how much they will participate. Successful F/OSS projects often rely upon one or more dedicated central figures. Examples of such central figures are Linus Torvalds in the Linux community and Guido van Rossum in the Python community. Module “owners” typically maintain control of the development of specific subsystems within larger projects, like Linux or Mozilla. The central figures and module owners are far fewer in number than the volunteer programmers and, in some contexts, may be financially supported through their projects.

Hybrid forms of F/OSS organization have emerged in which a core of participants may work for a traditional organization but their work assignment consists of working solely or in large part on the development of F/OSS software systems. Netscape used this hybrid model to kick-start Mozilla.org: many people already involved in the development of the commercial, closed and proprietary Netscape Web browser simply kept doing the same sort of work, as Netscape employees, in the context of a new F/OSS organization.

F/OSS development communities are sometimes described as networked or virtual organizations. Networked organizations are characterized as being flatter and less bureaucratic than traditional hierarchical / bureaucratic organizations. However, ethnographic research focused on the Python language community reveals a “strict, hierarchical model of production and social/organizational roles” (Sack, et al., 2004) (n.p.). The bug reports studied here show frequent (but not universal) use of an explicit two-step code review policy: each software change is supposed to undergo code review by a designated person with code review authority plus a second code review (“super
review”) by a more senior community member before the code change is integrated into the build.

Peoples’ skills and capabilities determine their role (meritocracy) as opposed to explicit designation of roles in traditional, bureaucratic software development organizations (Raymond, 1999). Bugzilla reifies roles through its structure and its implementation of roles: someone with the power to approve code changes has more power and stature in the community than someone who can only create new software; a programmer has more stature than someone who is allowed only to report problems.

Software engineering best practices, especially practices associated with normative SWPM process are a critical component of the structural context in the F/OSS community studied here. While the approach to solving particular software problems and managing particular bug reports is commonly discussed and negotiated, the general structure of the repository and its associated protocols (the coordination mechanism) and the overall process of report – triage – solve – verify is the same as that used in traditional, bureaucratic organizations (as reported in Carstensen, Sorensen & Tuikka (1995), Schmidt & Simone (1996), Crowston (1997), and Kajko-Mattsson (1999)). Code management processes, which include automated and human regression testing and code reviews, are also based on normative software engineering practices.

Other standard software practices are seldom employed, however. One oft-noted difference between F/OSS and traditional software development projects is the lack of formal requirements and design specifications in F/OSS projects, which are commonly used in traditional software development settings (Scacchi, 2002). F/OSS projects typically rely upon an informal, organic process referred to as “continuous design” (Gasser, et al, 2003; Sack et al., 2004).

Political / Ideological Context

This community is influenced by the political and ideological contexts, notably the ongoing tension between F/OSS advocates like Richard Stallman, Linus Torvalds, and
Eric Raymond and the sometimes antagonistic commercial interests like Microsoft. These issues are beyond the scope of this project, and are only briefly noted here.

1. The role of F/OSS as a “social good.”
2. Ethos of sharing and openness.
3. F/OSS as a political position (e.g., revolutionary idea or process; controversy over copyright).

Some commentators argue that the F/OSS development movement is a “social good” because it promotes the open exchange of intellectual capital, cooperation, communal learning, increased availability of high-quality and low cost software, improved productivity through the availability of better software, etc. The response of commercial software firms has varied widely, with some taking an active cooperative / co-opting approach (Netscape, IBM), others a neutral approach (Sun, Oracle supporting interoperability between their products and open source systems but showing less inclination to participate in F/OSS communities) and others an oppositional approach (Microsoft and SCO perceiving F/OSS as a threat to their particular commercial interests).  

Sharing and openness are central to the F/OSS development approach. Software is developed in the open (the source code is always available) and commentary, contributions and innovations are encouraged. Raymond (1999) compares the F/OSS process to the peer-review process used to evaluate scientific research, arguing that this open, above-board process ensures that the highest quality alternatives, both at a macro/systems level and at a micro/code level, are most likely to emerge as viable, sustainable projects and products. Volunteers are encouraged to participate in the

9 These relationships are not static and are subject to change. As of mid 2004, Sun, for example, is moving toward a more active position relative to F/OSS communities.
projects. As Sack et al. show (2004), it is possible for new participants in a community to move from the periphery to the center, based upon the quality of their participation. As the current project shows, more than the source code is shared in these projects. Test tools, bug report repositories and their content, and process documentation are also shared.

Finally, F/OSS is sometimes perceived as a revolutionary process or form of social organization. Raymond (1999) and Stallings are particularly known for advocating this points of view.

**Meso-level Contexts**

Meso-level contexts are those that span the community’s SWPM process in its entirety but do not encompass the entire community and all of its activities. One meso-level context identified during this project is the community’s bug report repository (an instance of the open source software problem management tool Bugzilla). This bug report repository is one component of the larger technological context discussed in the preceding section on structural contexts. The bug report repository is of central importance in this community and to this project because it is the nexus of the entire SWPM process.

The bug report repository is an example of what Schmidt and Simone (1996) call a coordination mechanism. A coordination mechanism is “a construct consisting of a coordinative protocol (an integrated set of procedures and conventions stipulating the articulation of interdependent distributed activities) on the one hand and on the other had an artifact (a permanent symbolic construct) in which the protocol is objectified)” [emphasis in the original] (p. 165-166).

The coordinative protocol is comprised of both elements internal to and external to the Bugzilla instance. The developers of Bugzilla itself have developed and published documents describing the best practices and the SWPM model upon which the design of Bugzilla is based (see, for example, Bugzilla.org (2004); Mozilla.org, bugs, (n.d.); and
Mozilla.org, *A Bug’s Life Cycle*, (n.d.)). On the other hand, like any software system, it is instantiated in a particular software image (a release) which provides limited flexibility in its adaptation and use within any particular community. Much of the coordinative protocol is embodied in the software itself. For example, there are particularly defined bug report status sequences and a limited number of legal status transition sequences.

Bugzilla as a tool, and this community’s instance of Bugzilla, prescribe aspects of the community’s SWPM process. The external conventions, as the community’s documents attest, further prescribe aspects of the process that are not coded explicitly into the bug report repository. The identification of duplicate bug reports is one example where the coordinative protocol is partly embodied in the bug report repository and partly embodied in the community’s conventions. Duplicate bug report submissions (that is, multiple bug reports created asynchronously by different people that describe the same phenomenon) are a challenge in any software development community because they can cause confusion through information overload, waste resources as multiple people address the same problem, and so on. The community has conventions, which it tries to enforce at the user interface level, to prevent the creation of duplicate bug reports. These include reminders and guidelines for submitters to search carefully for pre-existing bug reports that already describe the phenomenon they have encountered. The tool supports duplicate bug report identification by providing a powerful and flexible search interface, but is not designed to perform duplicate identification automatically and with a high level of accuracy. At present, the communicative protocol is insufficient as about one-third of all bug reports created are eventually marked as being duplicates of other bug reports.

The bug report repository also constrains activity by its implementation of roles and the allocation of capabilities for individuals based upon their roles. Individuals associated with the roles reporter, assignee and quality assurance contact have different capabilities regarding the creation and modification of bug reports which are enforced by the bug report repository itself. However, the role of code reviewer is not embodied in the tool; it is a convention understood by the coders and those individuals designated as code reviewers and code super-reviewers.
The control of access to the bug report repository is also based partly in the technology and partly in community convention. This community’s implementation of Bugzilla is open for querying and reading on the World Wide Web. In order to create a new bug report in this community, a person must register as a user of this instance of Bugzilla. This increases the cost (in effort) of creating a bug report and thus serves to reduce the number of spurious bug report submissions. Additional levels of access control exist, primarily to define community roles and provide appropriate levels of control to community members in those roles. Additional access controls are sometimes also imposed. For example, one bug report in the sample (BR 67) cannot be queried or read like all of the other bug reports in the sample and the vast majority of bug reports held in the repository. This community has a practice of imposing additional access control mechanisms for bug reports that describe security problems.

The requirement that only registered repository users can submit bug reports is part of the community’s effort to ensure the repository is used to record software problems and track the activities and information related to their resolution. Requiring registration is often a barrier to access (in terms of time and effort cost or risk of divulging personal information) that casual or curious users are unlikely to cross (Bishop et al, 2000). One practical effect is to inhibit the creation of bug reports that contain questions about how to use the software. For example, in bug report 375 from the sample analyzed in this project, we see a user reporting a problem and getting a response from a someone within the development part of the organization that expresses the opinion that the bug report is not a valid software problem. The reporter then counters with his rationale that the situation is a valid software problem (note the reporter’s sensitivity to the distinction between a “help desk” style question and a software problem):

------- Additional Comment #1 From <Commentator_1> 2002-11-03 10:23 -------
<Reporter>, strictly speaking, you’re asking for technical support. This database is only intended for bug reporting. You should ask this question in a forum intended for end-user support.

------- Additional Comment #2 From <Reporter> 2002-11-03 13:16 -------
Well, I thought about that. But then I decided that if something so innocent could cause such a serious result, it was a bug. The contents of the <component> should not be so "fragile."

Use of controlled vocabularies is another way in which the bug report repository constrains SWPM activity and process. Controlled vocabularies are used in ten fields within the bug report in this community (see discussion of the bug report structure in Chapter Four above). Some serve to define valid / legal bug report state sequences (discussed above) and others serve to classify bug reports along several dimensions (product; component; severity; priority; etc.). The controlled vocabulary fields provide simple means for community members to create collections of bug reports with various characteristics on the fly (e.g., all unresolved bug reports either reported by me or assigned to me). Variation in the number and purpose of fields with supporting controlled vocabularies would affect the community’s ability to create collections of bug reports dynamically. On the other hand, the search interface for the repository is complex: it exposes all of these fields in order to provide maximum flexibility for the user instead of making ease of use a top priority. Note also that while a limited set of specific values is available for any of these fields, their appropriate application and use in the context of particular bug reports is dependent upon some shared understanding of the SWPM work of the community and the relationship of SWPM to other community activities, like the development of new system functionality.

The bug report repository does not provide any support for some functions including detection of suspected duplicates, detection of informal or unrecognized relationships between bug reports (i.e., discovery of latent bug report networks and relationships), or formal completion of implied transitive relationships. For example consider a small network of duplicate BRs: if BR X is marked as a duplicate of BR Y and BR Y is marked as a duplicate of BR Z, BR X must also be a duplicate of BR Z. This repository does not automatically create the formal association between BR X and BR Z.

Finally, this bug report repository does not enforce the fixing of problems in the order implied by the dependency relationships created in the bug repository. If the lack of a code change to resolve the software problem represented by BR X blocks the creation
of a code change to resolve the software problem represented by BR Y, and this dependency is established in the bug report repository, the bug report repository cannot prevent someone for creating a patch for BR Y and marking that bug report resolved. Thus, the establishment of such relationships in the bug report repository functions, in reality, as documentation, or as a resource for action (Suchman, 1987), not as a strict rule or plan. Community members have the freedom and option to work around or ignore the various supported inter-bug report relationships.

**Micro-level Contexts**

Micro-level contexts are those contexts which reoccur many times in the data analyzed in this project. These contexts are frequently present within, as part of, or as objects of the field of work. This section focuses on three specific micro-level contexts: bug report networks, bug reports, and individual instances of negotiation. I estimate, based on the data in my sample, that there are tens of thousands of bug report networks created from the hundreds of thousands of bug reports, and I estimate that there are hundreds of thousands of instances of negotiation present in this community’s bug report repository. The examination of micro-level contexts causes the observational lens of the project to focus on greater detail than during the consideration of structural and meso-level contexts presented earlier in this chapter. The earlier section on structural contexts discussed constraints on the community as a whole and the previous section on meso-level contexts examined one critical constraining factor, the bug report repository, on the SWPM practices of the community.

**Bug Report Networks**

Bug report networks (BRNs) are virtual information objects created within the bug report repository by community members as they progressively assert various formal and informal relationships between individual bug reports. The formal relationships
supported by this community’s bug report repository are duplicate, blocks and depends on. Community members also assert a wide variety of informal relationships between bug reports, such as “see also” references (see Chapters Four and Five for additional discussion of BRNs). While an accurate estimate of the number of BRNs present in this repository is not yet available, the bug report sample analyzed in this project indicates that 65% of the bug reports contain at least one formal or informal reference to another bug report. It is likely that there are tens of thousands of bug report networks in this bug report repository.

The next several paragraphs discuss the ways in which BRNs constrain and affect the use and creation of information, and the community’s activities, processes and social order. As Figure 5-1 shows, even relatively simple BRNs contain bug reports with a wide variety of characteristics. Bug report A (BR-A) in that figure is a “meta” bug report, an “anchor” for a collection of bug reports created by a community member in order to co-locate bug reports sharing some characteristic of interest. Many of the fourteen other bug reports noted but not discussed in that example are members of other bug report networks, illustrating one of the interesting but as-yet unexplored aspects of bug report networks: their extent and interconnectedness in this repository.

Bug reports often shift contexts within BRNs. For example, bug report (BR-C) in Figure 5-1 was opened to report and represent a software problem. BR-C became the central bug report of a new, small BRN when BR-D was recognized as a duplicate bug report of BR-C the same day BR-C was opened. Thus BR-C existed in the context of a two-node BRN with a single duplicate relationship. About six weeks later BR-C itself was identified and marked as a duplicate of BR-B, changing its context from being the central bug report in a BRN to being a duplicate of another bug report, BR-B, which assumed its role as the central bug report describing the software problem. No further activity was noted in BR-C because it, by itself, was marked resolved and it was no longer the central bug report of any BRN. BR-C continued to exist in a new BRN context as a member of a larger, active BRN. BR-B was the central bug report representing the phenomenon described when BR-C was first created, and BR-B remained the locus of information.
about the community’s ongoing work to correct the underlying software problem. BR-C and BR-D also became part of the “meta” bug report network anchored by BR-A at the same time BR-C was marked as a duplicate of BR-B. It’s impossible to determine, but the information in BR-C remained available as a resource for the people responsible for resolving the software problem represented by BR-B.

BRNs, unlike individual bug reports, do not have a defined life cycle similar to the life cycle represented by the succession of status values associated with each individual bug report (see Table 4-4, Bug Report State Sequences). BRNs are not first-class information objects that undergo explicit management by the community even as the community creates and uses them. There is no way for a community member to mark a BRN “fixed” as she can with an individual bug report.

There are other limits to the support the bug report repository provides for the management of bug report networks. The repository does not, for example, support direct searching for BRNs: BRNs can only be located by first finding bug report that is a member. The repository provides only limited views, or summaries, of existing bug report networks in the form of a text-oriented dependency tree and graphical dependency displays. Neither duplicate relationships nor the many kinds of informal relationships are included in the summary displays.

The bug report repository does not use the formal relationships between individual bug reports to constrain the actions of people involved in the SWPM process. It is possible for a fix to be created for a software problem whose bug report indicates it is dependent upon a solution to the problem represented by the bug report “blocking” it. This is possible because of the distinction between software problems (phenomena) and bug reports (representations of phenomena) and the disconnect between the bug report repository and the code in use, which is the locus of the software problem. It seems technically infeasible (if not impossible) to create a software development system where the bug report repository and its states can prevent certain, but not all, changes in particular parts of the system’s code repository.
BRNs have a profound effect on social order, but, unlike bug reports, do not have associated formal roles like reporter, assignee and QAContact. The relationship between BRNs and social order is dynamic and is discussed in detail, using the BRN illustrated in Figure 5-1, in Chapter Eight, below.

The establishment and use of BRNs is also closely related to the community’s information practices. There is a clear relationship between the searching behavior of individuals and the establishment of “duplicate” relationships between bug reports. As noted above, the community has tried to establish guidelines and conventions to minimize the number of duplicate bug reports that are created. One critical part of the prevention of duplicates is the ability for reporters to search for and identify existing bug reports that already represent the phenomenon they have experienced. However, given the ambiguity of language and the complexity of the search interface, many duplicate bug reports are still created. Chapter Seven describes one 18-hour period in one day when the triage process identified 17 of 105 (16%) new bug reports as duplicates despite the preventative measures outlined above. One-third of the bug reports in the sample analyzed here were marked as duplicates of other bug reports.

There is also an important relationship between the evaluation and comparison of bug reports, the evidence they hold, and the establishment of both the formal and informal relationships used to construct BRNs. As noted previously, searching the bug report repository is a critical part of identifying duplicate bug reports and establishing “duplicate” relationships between bug reports. However, after performing searches aimed at retrieving possible and actual duplicates, the searcher has to rely upon his or her expertise to make decisions on whether there is a true duplicate relationship between the retrieved bug reports and the bug report under evaluation. Based upon the published protocol and recommendations for creating new bug reports, reporters also perform a similar search / evaluation pattern, the goal of which is to prevent the creation of unneeded, duplicate bug reports.

Finally, BRNs only constrain activity informally. For example, as noted in the discussion of the bug report repository above, dependency relationships do not prevent
work occurring on bug reports representing phenomena that are “dependent” upon the

correct of the “blocking” problem, whether that work is changing code in the code

repository or that work is modifying the bug report associated with the software problem.

One limitation of this project is that it is completely based upon the documentary
evidence present in bug reports and does not include any interaction with the participants
(e.g., via interview) to help understand the ways in which community members interpret
and employ the information conveyed by a BRN.

Bug Reports

Some features of the context of individual bug reports were already alluded to in
the discussion of the bug report network micro context (above). A summary of the
implications of the bug report context on information, activity and social order is
provided here.

A bug report contains limited kinds of information because the design of the bug
report (and the repository) limits the freedom individuals have to enter information. Most
of the data is in text format, although this repository’s attachment feature allows the
association of arbitrary forms of digitized information (e.g., other text objects; code
patches; screenshots). The repository’s time and identity stamping of bug report creation
and subsequent changes to the bug reports supports the need for accountability in the
SWPM process. The requirement that active participants involved in the SWPM process
must register to add or modify bug report data works to inhibit wholesale spamming of
the repository.

The mores of the community shape the information included in the bug reports
and the tenor of the contributions. The exchanges I have analyzed in the hundreds of bug
reports I read during the course of this project are overwhelmingly civil and on-topic
resulting in a high signal-to-noise ratio, in contrast to many other online communities
(e.g., newsgroups).
The bug report constrains activity, too, most notably in how the bug report’s formal status and resolution states support the specific coordination protocol (Schmidt & Simone, 1996) enacted by the community. The bug report also supports a specific social order, with formal roles (and associated system-enforced rights) defined for reporter, assignee and QAContact. Members of the cc: list are formally associated with a bug report, but do not have specific responsibilities. Other community members can comment on or passively monitor the changes made to the bug report, too. Chapter Eight discusses these roles in more detail.

Information practices related to the bug report context include:

- Searching to identify duplicate bug reports
- Evaluation and comparison of bug report information to identify and establish duplicate and dependency relationships between bug reports
- Evaluation of bug report information to confirm the validity of the report

It also seems likely that different individuals, playing different roles vis a vis a specific bug report, would find different parts of the bug report most important. The person triaging the bug report, for example, might be more interested in the instructions from the reporter on how to reproduce the problem and the information available to help search for and identify duplicates. The assignee might not be concerned about locating or identifying duplicates, but may be more sensitive to potential dependency relationships, the steps to recreate the problem, contextual information about the operating systems and platforms on which the problem does or does not occur, and the available detailed technical information like error messages and stack traces. A QAContact verifying the fix might ignore all but the steps to recreate the problem, considering all the information between the reporter’s recipe for recreating the problem as a black box that needn’t be opened in order to verify whether the problem is fixed or not. A code reviewer, however, is responsible for opening that black box and evaluating both whether the problem is fixed, not fixed or partially fixed, and the complex engineering evaluation of the quality of the fix. For example, the patch might fix the code,
but the patch might not be done in a way to prevent future problems, and ease future code maintenance.

**Negotiation Context**

SWPM is a kind of work in which some aspects are predictable (routine) and some are unpredictable (non-routine). The non-routine situations are interesting because the community must adapt to uncommon or uncertain conditions within a specific set of layered contexts (e.g., this software problem within this BRN represented in this bug report repository within this software development community, etc.). One of the basic social processes the community uses to deal with uncertainty is to engage in negotiation (see discussion on negotiation in Chapter Seven). The bug reports analyzed in this project indicate that negotiation is one of the most common basic social processes employed by community members to “get things done.” What is analyzed here is negotiation within the micro-level contexts, within individual bug reports or specific BRNs.

Strauss, in his work on negotiation as a social process (1978), identified the following properties of what he called *negotiation context*.

- The number of negotiators, their relative experience in negotiating, and whom they represent.
- Whether the negotiations are one-shot, repeated, sequential, serial, multiple, or linked.
- The relative balance of power exhibited by the respective parties in the negotiation itself.
- The nature of their [the parties in the negotiation] respective stakes in the negotiation.
- The visibility of the transactions to others; that is, their overt or covert characters.
- The number and complexity of the issues negotiated.
The clarity of legitimacy boundaries of the issues negotiated.

The options to avoiding or discontinuing negotiation; that is, the alternative modes of action perceived as available. (Strauss, 1978) (pp. 99-100)

Strauss uses the term negotiation context to describe the “properties entering very directly as conditions into the course of the negotiation itself” (p. 99). Bug reports, as dynamic artifacts, have different (and potentially unique) groups of people associated with them: the various reporters, assignees, commentators, and so on. Negotiation in bug reports represents negotiation of information, activity, and social order within each of these sub-communities. Negotiation at the bug report level is emergent, dynamic, and organic in the service of managing both bug reports (which may or may not represent problems) and the software problems that some bug reports represent. In contrast, the bug repository – this community’s coordination mechanism – changes far less frequently and within the context and constraints of the normative SWPM process described in Table 1-1. While Crowston’s and Schmidt and Simone’s theories can be applied to the analysis of structural context (e.g., coordination mechanisms), like a SWPM system, they cannot be applied to the study of individual instances of negotiation.

Negotiation may be particularly evident in this community because, as a F/OSS development community, it is relatively less hierarchical and less coercive than traditional software development organizations, which are usually embedded in bureaucratic organizations.

Three examples of negotiation are discussed in Chapter Seven, below. The eight properties of a negotiation context identified by Strauss are discussed in terms of those three examples.

- The number of negotiators, their relative experience in negotiating, and whom they represent:
  The first example includes two negotiators representing the reporter and a commentator. The second example includes four negotiators representing the
reporter, the assignee and two commentators. The third example includes five negotiators representing the reporter, two assignees and two commentators. In all three examples, the level of experience of the negotiators is difficult to determine based upon the evidence considered here\textsuperscript{10}.

- \textit{Whether the negotiations are one-shot, repeated, sequential, serial, multiple, or linked}: Examples one and two represent one-shot negotiations because a single issue is negotiated in the context of a single bug report. Example three is more complex, representing linked negotiations because four separate issues are being negotiated in the context of a single bug report (what is the best design?; who’s responsible for fixing this?; how should the bug report be managed? is this a duplicate bug report?).

- \textit{The relative balance of power exhibited by the respective parties in the negotiation itself}: The judgments presented here are subjective, but in example one, the two participants seem to have similar levels of expertise. Example two shows the first assignee being directed explicitly by the reporter; eventually, the reporter takes over responsibility for fixing the problem. In example three, the reporter, commentator 1, assignee 1 and assignee 2 seem to have similar levels of expertise. Commentator 2, however, uses his or her power to re-associate the bug report with a different component of the system to change who is assigned to fix the problem. Determination of the relative power of the negotiators is complex and likely determined by factors such as role within the community (e.g., a module owner likely has greater power than a QAContact or one-time reporter), reputation, etc.

- \textit{The nature of their respective stakes in the negotiation}: In general, the negotiations analyzed in this project are not adversarial; the parties to the negotiations are typically cooperating on solving problems, managing bug reports, and improving

\textsuperscript{10} Other methods, such as social network analysis, would be useful in determining each negotiator’s role and centrality within the community, or to evaluate the balance of power between negotiators.
the software. However, there are clearly cases where negotiation, while cooperative in nature, fails to result in the setting of the bug report status to “resolved,” much less the correction of an acknowledged software problem. The stakes here seem to be relatively low: influencing design decisions, priorities, etc., rather than “winning” as may be the case in many other kinds of negotiation (e.g., salary or contract negotiation).

- The visibility of the transactions to others; that is, their overt or covert character: all of the negotiations analyzed here are presented in a public forum, although other negotiations or other parts of visible negotiations may occur via other channels, some of which, like personal e-mail, chat, instant messaging, may be unknown to the community at large. Negotiations whose subject is the determination of a “best” design are sometimes long in duration, and negotiations regarding software or interface design issues are often conducted in multiple forums, like newsgroups and specific bug reports.

- Number and complexity of the issues negotiated: Thirteen topics or issues are frequently negotiated in this community’s bug reports (see Chapter Seven for more details). The issues “Is there a problem?” and “What is the best design?” frequently result in multiple-turn exchanges recorded in the bug reports, reflecting the potential complexity and contentiousness of these issues. The third example of negotiation discussed in Chapter Seven also reveals the complexity that occurs when multiple issues are negotiated within a single bug report. About 27% of the bug reports in the sample analyzed here contain evidence of negotiation on two or more topics.

- The clarity of legitimacy boundaries of the issues negotiated: The documentation guiding the SWPM in this community does not specify guidelines for negotiation. The issues and the ways in which this community negotiates appear to be emergent and organic. Yet, as discussed in Chapter Seven, the issues negotiated are bounded and clearly focused on the management of software problems and bug
reports. Determining the roots of how this clarity came to hold is beyond this scope of this project.

- *The options to avoiding or discontinuing negotiations; that is, the alternative modes of action perceived as available.* Egger & Wagner (1992) refer to a tension between “leadership and democracy” in instances of negotiation within teams. In the relatively flat, non-coercive context of a F/OSS development project, this tension is also sometimes apparent. However, this project does not focus on identifying the roles or relative authority of particular individuals in the community, so it is not possible to state with any certainty whether the imposition of authority is used as a means of terminating negotiation or as an alternative to negotiation. There is a sense in some cases that an individual can exert influence in the outcome of the negotiation by taking specific action. There is also some evidence that, especially in those bug reports that lack resolution or display no activity for a long period of time (bug reports that seem to get “lost” or “forgotten”), the community cannot reach consensus or take a decision.

The preceding discussion shows that Strauss’ eight properties of instances of negotiation provide a useful starting point for analysis and consideration of specific negotiation contexts. Further analysis of negotiation contexts in this and other settings is warranted to better understand the relationships between information practices and this basic social process.

**Conclusion**

Talk about idea that the repository is not over-specified: it is simple and amenable to adaptations like meta-bug reports and “violation” of the soft dependency constraints. Conventions about code reviews and the related signals are not enforced either.

This chapter presented an analysis and interpretation of the layers of context that affect the information practices of this distributed software development community. I selected those contexts that affect the community in its entirety as structural contexts. I
identified as meso-level contexts those that affect the SWPM process as a whole. Finally, I categorized contexts that related closely to the objects of the field of work, such as bug reports, instances of negotiation, and bug report networks, as micro-level contexts. The micro-level contexts reoccur multiple times: there are hundreds of thousands of bug reports in this community’s bug report repository; I estimate that there are tens of thousands of bug report networks and hundreds of thousands of instances of negotiation. Allocation of contexts to the categories structural, meso-level and micro-level is to some extent arbitrary and my decisions on classification can certainly be debated. Other important contextual layers may exist, but are not part of this project (e.g., a software problem context as distinct from the bug report or bug report network contexts). This chapter raised issues regarding structural context, and discussed in more detail findings regarding specific meso- and micro-level contexts.

Consideration of context, while difficult, must be addressed when analyzing the information practices of people in realistic, complex social settings. SWPM in the F/OSS development community studied here is a (globally) distributed collective practice. Imagine how this community’s SWPM process would work if, instead of relying on a Web-accessible relational database management system-based repository that uses frequent asynchronous computer mediated communication (e-mail notifications), the repository were paper based as described by Carstensen, Sørensen, and Tuikka (1995). How would community members be notified of new problems to be triaged? How would assignees learn that they were responsible for particular bug reports? How would community members not formally associated with the bug report as reporter, assignee, QAContact provide their expertise, contribute to negotiation, etc., in support of the sensemaking process? It’s clear that this community could not operate in nearly as efficient a way under those conditions (remember that the community can create on the order of 100-200 new bug reports per day).

This project does not provide definitive guidance for researchers on how context should be taken into account. Context is a complex methodological issue that is only beginning to be addressed by researchers in information science. The contribution here is
to provide evidence of the complexity of context within distributed organizations, and to promote the view that decomposing context using a layered model can help make the complexity tractable. In this community, the structural, meso- and micro-levels of context comprise a grand context (Pettigrew, 1999) from which insights into human information behavior may, with additional analysis, be drawn.

We can obtain a clearer understanding of how these overlapping and mutually influencing layers of context affect information practices by conducting additional comparative studies of work in a variety of settings. For example, the SWPM practices of other F/OSS development communities could be compared to this community to determine similarities and differences in between each community's information practices, processes, activities and social order. Large communities, like the one studied here, could be compared to smaller communities. Traditional software development communities could be compared to F/OSS communities. Communities engaged in activities other than software development could be compared to various F/OSS and traditional software development communities (thus contributing to the development of theory).
CHAPTER SEVEN: PROCESS IN DISTRIBUTED WORK

Introduction

This chapter presents the findings from this research project related to process in distributed work. The processes employed by a distributed work community are a primary determinant of both the nature and the quality of its outcomes and outputs. The details of the community's processes also are influenced by contextual factors, such as the level of process accountability required and the systems used to support the processes. Processes mutually constitute information: information as input to, as impetus for, and as output of processes. Processes also constitute the social order: when a new bug report is triaged, for example, particular community members are brought together in a cooperative ensemble whose goal is the expeditious resolution of the suspected problem.

Analysis of the bug reports created by the F/OSS development community studied here reveal a variety of processes including negotiation, the triage process, distinctions in process between the management of bug reports and the management of software problems, community reenactment, and voting. The focus of this chapter is on two processes: the software problem management (SWPM) process in its entirety, particularly the phase of the SWPM process known as triage, and negotiation. The SWPM process is one component of a larger, more complex process, the software development process. Both the software development process and its sub-process, the SWPM process, can be thought of as being large, complex business or engineering processes. SWPM has typically been analyzed as a business or engineering process in the few cases where it has been the subject of research in the past. The SWPM process is shown here to be more complex in this community than the existing descriptive accounts. The SWPM manage as a whole can be cast as a sensemaking process. Triage is a particularly interesting and critical part of the process and has not been described in depth before now. Negotiation is a basic social process employed in a variety of ways in all social settings. Its role in the SWPM process will be described in detail in this chapter.
Particularly surprising is the extent to which negotiation is present in the bug reports and the variety and intensity of negotiations conducted by this community in support of the SWPM process.

This chapter argues that the processes and sub-processes of SWPM in this community can be conceptualized as nested processes, and that consideration of the relationships of these processes to information practices, activity, information and social order sheds light on the way distributed work is performed in this community. The SWPM process is part of a larger software development process, but the SWPM process is the largest-scale process layer considered in the context of this study. Here, the SWPM process is subdivided into significant phases including problem identification and reporting; bug report triage; expert analysis, fix development, testing, deployment; and verification and closure (see Table 7-1).

Within each phase (e.g., expert analysis, fix development, testing, and deployment), finer-grained processes like collective debugging and individual debugging can be identified. Negotiation is a basic social process that is employed in both fine-grained processes like collective debugging and larger-grained processes like triage.

The findings presented in this chapter are grounded in the data analyzed in this project. The software development as a whole is only mentioned as the context within which the SWPM process is embedded, and is not considered in detail here. SWPM as an important component of the software development process, and the phases of SWPM (e.g., problem identification and reporting; bug report triage; expert analysis, fix development, testing, and deployment; bug report verification and closure) are analyzed primarily from the point of view as sensemaking processes (Weick, 1995). The findings from the analysis of the bug reports in the sample are presented as they relate to each of the four phases of the SWPM process and how they provide insights into the information practices (e.g., searching within the bug report repository; evaluation of information), activity, and social order of this distributed community. Negotiation, a basic social process, is described in terms of where and how it is employed within the different phases of the SWPM process. Collective debugging was a focus of this project to the
extent that its examination addresses issues such as distributed information practices and as a context for instances of negotiation. Collective debugging and individual debugging as sensemaking processes have been the subject of significant research in the past, as noted in Chapter Two. Identification of these processes and their relationships is valuable because it simultaneously signals the limitations of this particular study and suggests how future research projects might be designed in order to provide additional insight into the relationships between nested processes and sub-processes, and between process and activity, social order, information and information practices.

Figure 7-1 Processes and Sub-Processes in Distributed Software Problem Management
The Software Problem Management Process

This part of the chapter summarizes findings that enrich and extend the normative SWPM process model presented in Chapter One, providing an end-to-end, temporal overview of SWPM. Weick’s sensemaking theory (Weick, 1995) is used to unify the discussion of the SWPM process. The normative process is re-presented and re-interpreted based upon the findings from this project. The four major phases of the SWPM process are discussed: problem identification and reporting; bug report triage; expert analysis, fixing, testing and deployment; and bug report verification and closure.

The SWPM processes and techniques used by this F/OSS development community are based upon the SWPM model commonly used by medium- to large-scale software development organizations (Carstensen, Sørensen & Tuikka, 1995; Schmidt & Simone, 1996; Crowston, 1997; Kajko-Mattsson, 1999)). Table 7-1, based upon Table 1-1, summarizes the normative model and highlights some notable findings from the current project. While the normative model divided the SWPM process into six distinct steps, the revised model divides the SWPM into four phases as shown in Table 7-1. Table 7-1 shows both the four phases and the six steps in order help the reader follow the discussion that follows. The table also summarizes some of the findings from this project that enrich our understanding of the SWPM process.

Table 7-1 Normative and F/OSS Software Problem Management Processes

<table>
<thead>
<tr>
<th>Software Problem Management Process</th>
<th>Normative Process</th>
<th>Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase One: Problem Identification and Reporting</td>
<td>Step 1. Problem Identification</td>
<td>Detailed consideration of the actual encounter by a person with the problematic phenomena is beyond the reach of the bug report text. Based upon reading bug reports and their descriptions of the phenomena encountered, and the information provided by the reporter regarding the contexts of these encounters, it is apparent that there are many situations that lead to initial encounters with these phenomena.</td>
</tr>
<tr>
<td>Someone using the software notices an anomaly or mistake; identification can occur during normal use or during software testing; actors identifying problems may be testers (Carstensen, Sørensen &amp; Tuikka, 1995; Crowston, 1997) or customers / users (Crowston, 1997).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phase One: Problem Identification and Reporting</td>
<td>Findings</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td><strong>Step 2. Problem Reporting</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Someone creates a bug report - a</td>
<td>Many bug reports do not represent software problems:</td>
<td></td>
</tr>
<tr>
<td>representation of the bug - to enable the</td>
<td>design ideas; requests for enhancements; parties; tests of the</td>
<td></td>
</tr>
<tr>
<td>bug and follow-up activities to be tracked</td>
<td>bug report repository; mistakes (e.g., accidental double</td>
<td></td>
</tr>
<tr>
<td>through a bureaucratic process; actors</td>
<td>submissions). Only the phenomena represented by bug</td>
<td></td>
</tr>
<tr>
<td>creating the bug report may be testers</td>
<td>reports can be input into the rest of the SWPM process.</td>
<td></td>
</tr>
<tr>
<td>(Carstensen, Sørensen &amp; Tuikka, 1995) or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intermediary such as a help desk on behalf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>of the customer or user (Crowston, 1995).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Phase Two: Bug Report Triage**

Someone takes action to begin the process of identifying and fixing the bug. An attempt to recreate the problem is made. The priority of the bug is assessed (triage); the proper person or organization to work on the bug is identified (assignment). In Carstensen, Sørensen & Tuikka’s study (1995), priority and assignment are handled by a group of software designers whose role is the management of software problems. In Crowston’s study (1997), set in a larger organization, problem recreation and prioritization is done by “marketing engineers” and assignment by “programming managers.”

Reducing the number of bug reports that need the attention of the community is a high priority. Identification of duplicates is the number one strategy for reducing the number of bug reports needing additional attention; marking others as “invalid,” “won’t fix,” “works for me” are other common strategies for reducing the size of the workload. The relatively small number of high priority problems are identified and hopefully fixed almost immediately. Standard default assignees, based on the controlled vocabulary fields within the bug report, like product, etc., simplify the triage process. E-mail notifications are sent automatically, helping ensure that someone is quickly made aware of each new bug report. Effective and efficient notification is important to ensure fast response to high priority problems. The triage phase is influenced by the technical context as embodied by the bug report repository.

**Phase Three: Expert Analysis, Fixing, Testing, and Deployment**

The assignee investigates the bug, determines its cause, evaluates repair options, consults other experts, coordinates the work of multiple experts, etc. The assignee modifies the software to resolve the bug and tests the modification. The modification, or fix, is deployed if testing is successful. Software designers (Carstensen, Sørensen & Tuikka, 1995) or software engineers (Crowston, 1997) usually perform this task.

Collective debugging, individual debugging, voting and negotiation are some of the sensemaking, community specific, and basic social practices employed in this phase of the SWPM process. Previous research has focused on how individuals and groups debug problems; other research has focused on the software engineering aspects of change management and software maintenance. See the preceding chapters for details on the information, activity, and social order related to this part of the process.
### Phase One: Problem Identification and Reporting

This section covers the period between the human encounter with a problematic phenomenon and the creation of a bug report that represents that phenomenon, inclusive. This phase is divided into two steps, problem identification and problem reporting. This section also emphasizes important distinctions between (1) the existence of a problem (a phenomenon) (2) the human encounter with the phenomenon and (3) the creation of a bug report that represents the phenomenon. These distinctions are important both to emphasize the key distinction between bug reports and the phenomena they represent and to point out other distinctions between (1) problems encountered and not encountered and (2) encountered problems represented by bug reports and encountered problems not represented by bug reports. Finally, a number of bug reports contain language indicating that they are perceived to be of a certain type; this section includes a section describing the distinct bug report types found in the sample.

---

**Table 7-1, cont.**

<table>
<thead>
<tr>
<th>Software Problem Management Process</th>
<th>Normative Process</th>
<th>Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Phase Four: Bug Report Verification and Closure</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Step 1. Fix Verification</strong></td>
<td>Someone (who usually plays a quality assurance role) other than the assignee verifies that the bug has been corrected. The platform master (the person responsible for the software build containing the fix) (Carstensen, Sørensen &amp; Tuikka, 1995) or the integration team (Crowston, 1997) may take on this role.</td>
<td>Verification is sometimes done by the reporter or assignee; in most cases, the bug report is left alone after the status is set to “resolved.” This apparent lack of consistency may be a product of the relatively informal organizational structure in place in this community. One organization with which I was affiliated was seriously committed to ensuring that bug reports moved properly through the verification and closure steps: bug report status was reviewed weekly and all bug reports are expected to move to a “closed” state.</td>
</tr>
<tr>
<td><strong>Step 2. Problem Closure</strong></td>
<td>Someone marks the bug report “closed:” the process is complete. In Carstensen, Sørensen &amp; Tuikka (1995), this is the responsibility of the central file manager; in Crowston (1997), this is the software engineer’s responsibility (Crowston, 1997).</td>
<td>While “closed” is a valid status in the bug report repository used by the community studied here, bug reports are almost always left untouched after the resolution has been set to “verified.” Less than 1% of the bug reports in the repository have their status set to “closed.”</td>
</tr>
</tbody>
</table>
Problem identification is the first step in the first phase of the SWPM process. It is more convenient to consider these two steps as unit for the purposes of this discussion, which begins by examining the distinctions between the phenomenon (the problem), the encounter and the act of creating an information object to represent the phenomenon, a bug report. Table 7-2 summarizes the combinations of whether a problem exists or not; whether a problem is encountered or not; and whether a bug report representing the problem is created or not.

Row 1 represents healthy software. Row 2 is unlikely to occur and a bug report created under these conditions would likely be closed quickly and marked “invalid” or “works for me.” Row 3 represents an unreported encounter of a phenomenon that would not be accepted by the community as a valid problem if reported (row 4). Row 4 represents a common situation in the community studied here: someone reports their problematic encounter but it is not accepted as a valid software problem by the community because it is either (1) invalid (e.g., BR 99 where the reporter claims that the system is not compliant with one of the standards specifications, but the system as implemented is determined to be in compliance), (2) a phenomenon that won’t be fixed (e.g., BR 95 where a user interface enhancement is proposed, but is shot down in favor of the existing, current implementation), or (3) a phenomenon that cannot be recreated after the bug report is created (e.g., BR 100 where a user reports a system hang, but neither he nor the assignee can recreate it again later).

Rows 5 and 7 represent situations where a software problem exists, but it is never reported, so the SWPM process is never invoked. (Like the situation where a tree falls in the forest but no one is near enough to hear it: does the falling tree then make a sound? In the context of this project, the answer is “no.”) Row 6 is of little interest because, given the current technological context, bug reports cannot be created without human intervention. Row 8 represents the most commonly occurring situation: a valid software problem is encountered and reported by a member of the community. In sum, the

11 For the purpose of this project, the submission of a “talkback” report, increasingly common in contemporary software systems, is not equivalent to creation of a formal bug report.
combinations of most interest to us are listed in rows 4 and 8 because these are the only reasonable situations that lead to the creation of bug reports, and this research is based upon the existence of bug reports.

Table 7-2 Phenomenon / Encounter / Bug Report Combinations

<table>
<thead>
<tr>
<th></th>
<th>SW Problem Exists</th>
<th>SW Problem Encountered by Human</th>
<th>SW Problem Represented by Bug Report</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Working, healthy software.</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Unlikely for BR to be created in this situation; perhaps a mistake.</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Unreported encounter (if BR created, would likely be marked invalid, won’t fix, or works for me).</td>
</tr>
<tr>
<td>4</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>BRs like this are likely to be marked “invalid,” “won’t fix,” or “works for me.” Possibly a “request for enhancement” situation.</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Un-encountered and unreported problem; a latent, undiscovered software problem.</td>
</tr>
<tr>
<td>6</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Unlikely situation: software problems can’t create BRs without a human intermediary in this community (although it would be possible to design in capability for automatic notification of software faults to occur without any human intervention).</td>
</tr>
<tr>
<td>7</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>A valid software problem is encountered, but not formally reported; a second type of latent software problem as far as the software developer community is concerned.</td>
</tr>
<tr>
<td>8</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>A valid software problem, encountered, and reported.</td>
</tr>
</tbody>
</table>

In the Table 7-2 (above), note that, even though the “Yeses” and “Nos” in the first column may imply certainty, the data from the bug report sample shows that the community is often uncertain about whether phenomena reported in a bug report is a software problem or not. The question of whether a bug report actually represents a problem is one of the most frequently negotiated issues in this community as noted in
Chapter Seven. Another example where existence of a problem is debatable is the case of a request for enhancement: these requests are usually considered and talked about as not being software problems. It could be argued that missing or partial functionality represents a software design problem, but software design problems are not typically called bugs or software problems, or represented by bug reports.

The distinction between encountered problems represented by bug reports and encountered problems not represented by bug reports must be acknowledged because a SWPM process modeled upon the normative SWPM described here cannot “manage,” or be used to manage, either un-encountered problems or encountered bug unreported problems. Thus there are potentially large sets of software problems that are beyond the purview of any SWPM process based upon this model.

Problem reporting, step two in the first phase of the SWPM process, represents the point where the encountered phenomena is transformed into a first-class information object, a bug report. Creation of a bug report is a critical step because the rest of the SWPM process can only occur when a bug report exists. However, not all bug reports are created with the intention that they represent software problems. Bug reports analyzed in this project reveal that people create bug reports for other reasons including:

- Managing / grouping / collocating other bug reports (meta / tracking bug reports)
- Testing the repository
- By mistake (e.g., duplicate submission)
- Documenting an idea
- Requesting an enhancement
- Planning a party

The bug reports analyzed here also reveal some interesting uses of language and conventions that sometimes reveal the intentions of the creator of the bug report. In the
following section, some of the different bug report types that have been identified are discussed.

**Evangelism:** While most bug reports reflect problems identified in the software developed by the F/OSS development community studied here, there are occasions when problems are identified that are determined to be caused by an external entity. Thinking about software development generally, these situations may occur more frequently when software systems interface or interact with external systems developed by other communities or organizations. In this F/OSS community, one common guiding principal is adherence to open, public standards for computing and communications (noted as part of the technical context in Chapter Six). Evangelism bug reports indicate to this F/OSS community that the resolution to the problem cannot be provided from within this community (i.e., by modifying the community’s software) but must be supplied by convincing an external organization to make some sort of change (e.g., to conform to an established standard). This community uses the term evangelism in the ironic manner common in technology companies and communities: the community is reaching out to external people to encourage them to change their behavior in a way that is aligned with the goals of this community. In some cases, this community even proposes and supplies specific corrections to the external organization in order to achieve the community’s goals, which include both fixing immediate problems and also educating others about how to “do the right thing.” Seven of the 385 bug reports in the sample (2%) were bug reports of this type.

- BR 1: this bug report was open and active for more than 2.5 years as the organization worked to change the habits of developers outside of the organization. This was also an effort to encourage conformance to emerging technical standards.
- BR 138: is a case where this community contacts an external organization and gets them to modify their software, yet there is negotiation within the
community about whether changes to their own software should be made or not in case similar situations arise in the future.

- BR 175: this is a more straightforward case of encouraging an external organization to comply with standards, and succeeding.

**Reminder:** A bug report created as a reminder about a task or class of similar tasks that need to be performed. Two examples of a reminder bug report were found in the sample analyzed for this project. (Note that both examples of reminder bug reports occurred relatively early in the chronologically arranged bug report sample (BR 1 is the bug report in the sample with the earliest creation date; BR 385 has the latest creation date). One interesting implication of this, as yet unverified, is that the information in the bug report may reveal changes in the use of language, the organic evolution of the community’s SWPM process, or the growth of the development community itself from a relatively small, close-knit group of people to a larger, more heterogeneous group of people.)

- BR 10: the reporter opens the bug report with the description:

  `<reporter> Need to update the RDF parser and associated files to use the latest RDF prefix. It'd be nice if we could also handle "equivalent" RDF.`

  In BR 10, the reporter creates the bug report to remind himself of what to do.

- In BR 16, the reporter creates a reminder bug report to remind someone else of what to do, as a means of ensuring that a commitment by someone else is met.

  From the bug report description:

  `<reporter> Reminder bug for <assignee> to remove <component_name> from xpwidgets if it is indeed obsolete.`

  **Change notice:** BR 14 was created in order to notify the community about a change that was already made. Change notices are commonly part of software change
control processes, but this community’s bug report repository is not designed to be, nor is it normally used as, a change control system.

<reporter> In order to get <system component> to load the Photon widget/gfx libs instead of GTK I had to make this change to the Makefile.in. It seems reasonable enough to me considering its a patch layered on a hack (<identifier removed>, no offense intended)

There is also variety in the types of bug reports created as a result of encounters with problems during testing. These terms have been found in this project’s bug report sample:

- **Regression**: a bug that was not present in a previous build or release that now manifests. Examples are BRs 97, 114, 142, 233 and 238. There were twenty bug reports in the sample marked as regressions.

- **Stopper**: apparently a serious bug that impedes systems development progress. This term seems to disappear from use. It occurs in BRs 13, 17, 23, 27, 36 and 184. The bug reports containing this term either make the claim that the problem represented by the bug report either is or is not a “stopper.”

- **Smoketest**: a daily, automated test process used to “smoke out” serious problems with the new daily build (presumably, this name refers to the old saying “where there’s smoke, there’s fire). Bug reports using this terminology are either reporting problems uncovered during the smoketest or related to execution of the automated smoketest procedures. BRs 13, 100 and 200 contain this term.

- **Meta bug report**: A BR filed in order to help manage the work on a collection of BRs that are related in some way (e.g., in same code or functional area; due to be fixed by a common code build point; assigned to a single person, etc.) a.k.a Tracker or tracking bug. Bug reports in the sample that are meta bug reports are BRs 5, 75
and 220. Bug reports referring to other bug reports functioning as meta bug reports are BRs 17, 23, 236 and 261.

**Phase Two: Triage**

*Triage,* “the sorting of and allocation of treatment to patients and especially battle and disaster victims according to a system of priorities designed to maximize the number of survivors” (MerriamWebster.com, 2004) is the performed by the community following the creation of a bug report. Given the creation of a new bug report, the F/OSS development community tries to address the issue raised in the bug report quickly, efficiently and correctly. The community has limited resources (people, time, and attention) and positions its SWPM process as a means of managing software quality while maximizing the community’s problem solving resources. This section addresses how the community performs triage, which kinds of information are useful in triage, the objectives of the triage process, the strategies used to support effective triage, and the boundaries of triage phase within the larger SWPM process.

Triage is itself a relatively large-scale process that is primarily concerned with the management of the bug reports themselves rather than the problems they may represent. This was apparent from the results of a small investigation I performed on the day I was writing this part of the document, I checked the latest bug reports and saw that 105 new bug reports had been opened in the first 18 hours of the day. Out of those 105 new bug reports, the following resolutions were already identified:

About 27% (28) of the new bug reports had been somehow removed from the workload in the first 16 hours of the day through application of five different strategies: (1) mark as duplicate, (2) invalid, (3) won’t fix, or (4) works for me; or, (5) create a patch that fixes the reported problem. Only 3 of these bug reports had been “fixed” by changing the software; 24% of these 105 bug reports were effectively, if not completely or permanently, removed from the field of work almost immediately, without the effort of designing, coding, testing and deploying a code change. We can’t say that these bug
reports are completely removed from the field of work because bug reports marked duplicate may contain descriptive information or have associated attachments that remain useful for resolving the underlying problem. We can’t say they are permanently removed because any bug report marked resolved may be re-opened.

<table>
<thead>
<tr>
<th>Disposition of Bug Report</th>
<th>Count</th>
<th>Percentage of BRs</th>
<th>Cumulative Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marked “works for me”</td>
<td>1</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>Marked “wont’ fix”</td>
<td>1</td>
<td>1%</td>
<td>2%</td>
</tr>
<tr>
<td>Marked “fixed”</td>
<td>3</td>
<td>3%</td>
<td>5%</td>
</tr>
<tr>
<td>Marked “invalid”</td>
<td>6</td>
<td>6%</td>
<td>11%</td>
</tr>
<tr>
<td>Marked “duplicate”</td>
<td>17</td>
<td>16%</td>
<td>27%</td>
</tr>
<tr>
<td>Remaining to be worked on</td>
<td>77</td>
<td>73%</td>
<td>100%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>105</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is difficult to determine strict criteria for identifying the end of the triage and the beginning of the next phase. It might be possible to set a more or less arbitrary time limit on the triage period (for example, the first 24 hours following the creation of the bug report). Or, we might define the triage period as the first continuous / contiguous period of activity: when 24 hours passes without activity following the creation of a bug report, the triage phase is over. Or, we might define the end of triage as the transition of a bug report into one or more particular formal states (as represented by the “status” field). Or, we might simply accept that the end of triage is hard to identify, and that triage may reoccur during the expert analysis phase.

In the following examples, identities of individuals are changed and some system-related names are disguised (framed by angle brackets: thus Microsoft Windows becomes <Commercial Operating System>).

**Triage Example 7-1, Bug Report 321**: BR 321 is an example of triage working quickly and efficiently, with this critical severity bug report resolved as a duplicate; the default assignee (assignee_1) notes he is able to reproduce the problem, presents resulting debug / stack trace information, and notes that he was able to identify a preexisting bug
report. Assignee_1 makes a reference to searching for duplicates; assignee_1 also had to evaluate the information he or she had created when trying to reproduce the problem to the information contained in bug report found during the search for duplicates. There are no instances of negotiation.

Day one, time +01:29, comment #1:

<assignee_1>: I could reproduce this with my debug build:
First I got a few assertions:
<assertions removed>
and a crash:
<stack trace removed>
and after searching I found bug <master_bug_report_number>
*** This bug has been marked as a duplicate of <master_bug_report_number> ***

Triage Example 7-2, Bug Report 90: BR 90 represents another critical severity problem that causes the system to crash. This example illustrates the triage process for a severe problem where there is early uncertainty about where within the system the cause of the problem lies and, therefore, who should be responsible for fixing the problem. In the following edited presentation of highlights from the bug report, the comments are shown with relative timestamps, beginning with time 00:00, the time the bug report is created. Subsequent timestamps are given as + hh:mm to indicate the time elapsed since the bug report was created.

Bug report created by reporter on day one, time 00:00. The reporter provides a thorough description of the problem, the conditions under which it occurs, behavioral differences on different operating systems, build number, steps to reproduce the problem, an educated evaluation of who is responsible for fixing the problem, and stack traces relevant to the crash.
Day one, time +00:05, comment #1:

Assignee_1: looks like <someone>'s progress dialog box.

Add assignee_1 to CC: list.
Change assignedTo from <assignee_1> to <assignee_2>.

Note that the assignee re-assigns the bug report almost immediately. Allocating problems to the most appropriate party is one of the primary goals of triage. This change of assignee also indicates negotiation on the topic of “who’s responsible for fixing this?” The writer of comment #1 also adds himself/herself to the CC: list in order to monitor progress on problem, and because he/she is no longer the assignee, expanding the dynamic cooperative ensemble interested in this bug report.

Day one, time +00:33, comment #2:

Assignee_2: Yeah, looks like <assignee_2>'s file progress dialog box is exposing yet another bug in <subsystem_1> :-). Seriously, I don't think it's <subsystem_1> this time :-). Something is horked in <subsystem_2>. I guess I'll have to figure out what's happening and route this to the appropriate party. I've heard a rumor of another odd <subsystem_2> window loading error that <CC: list member_2> is working on; maybe it's all the same bug.

Change status from NEW to ASSIGNED. <Assignee_2 accepts the bug report.>

Note that assignee 2, besides accepting the assignment of the bug report, is beginning Phase Three, expert analysis, etc., by hypothesizing on which subsystem might be causing the problem. In comments #3-4 assignee_2 adds another stack trace and console output, compiling more evidence to support the expert analysis.

Day one, time +00:36, comment #3:

Assignee 2: Here's the call stack when I hit the "Components not in scope" assertion:

<stack trace removed>
Day one, time +00:39, comment #4:

Assignee_2: And here's the console at the same point:

<console log removed>

Day one, time +00:52, comment #5:

Assignee_1: per <assignee_2>'s comments above cc'ng the big <CC: list member_2>!

Add CC: list member_2 to CC: list.

Note another expansion of the dynamic cooperative ensemble.

Day one, time +01:07, comment #6:

Assignee_2: Here's my theory:

nsStreamTransfer::SelectFile has been called and has returned (we see its output on the console but it isn't on the call stack). That function is called from nsStreamTransfer::SelectFileAndTransferLocation, which has also completed (it ain't on the stack, either). That function opens a new dialog which kicks off a complex sequence of asynchronous events, culminating in completion of the document load and invocation of the dialogs onload handler.

That has happened (I believe; see the stack entries in the neighborhood of GlobalWindowImpl::HandleDOMEvent) and the onload handler is executing (presumably). That code does this: "data.observer = observer" where data is an xpconnect wrapper for a nsStreamXferOp object. So it is trying to call, via xpconnect nsStreamXferOp::SetObserver.

Note that there is no reference to "Components" in downloadProgress.js. My theory is that xpconnect references Components when trying to call the nsStreamXferOp method (either SetObserver or Start). More likely the former since it actually has an argument.

Why this fails is the question at hand. I'm reassigning to the XPConnect component. Note that I've got a slightly modified nsStreamTransfer/nsStreamXferOp implementation (but identical, for all intents and purposes) up to this point. If you encounter a different scenario and need assistance understanding the stream-xfer aspects of it, please don't hesitate to ask.

Remove <CC: list member_2> from CC: list. <this was not intentional>
Change assignedTo from <assignee_2> to <assignee_3>.
Change status from assigned to new.
Change component from <component_1> to <component_2>
Change QAContact from <QAContact_1> to <QAContact_2>

In comment #6, assignee_2 is providing a relatively complex provisional explanation of the cause of the problem. I interpret this as negation on the topic of “Is bug’s cause determined?” Assignee_2 reallocates this bug report to another component
within the system thus changing both the assignee and QAContact, again expanding the
dynamic cooperative ensemble. (Note that CC: list member_2 is removed by mistake,
only to be re-added one minute later.)

Day one, time +01:08, comment #7:

Assignee_2: Adding <CC: list member_2> back to cc: list (lost due to mid-air
collision).

Add CC: list member_2 to CC: list.

In about one hour, the size of the dynamic cooperating ensemble changed from
three (reporter, assignee and QAContact) to seven (reporter, three assignees, two
QAContacts and one additional person on the CC: list). Two instances of negotiation
occurred, and the overlap between phases two and three are clear to see. Information
practices in support of triage included data collection, data organization, data
interpretation, and hypothesizing. We can’t tell with certainty if anyone performed
searches to locate duplicates, but we see in comment #2 that assignee_2’s memory and
experience identified a possible relationship with other development work in another part
of the system.

**Phase Three: Expert Analysis, Fixing, Testing, and Deployment**

This phase has been the subject of attention of researchers in the past. Some of
the literature about collective and individual debugging covered in Chapter Two. The
information practices of the community members as they work a bug report through this
phase is within the scope of this project. Common information practices observed during
this phase include evaluation of evidence contained in the bug report. Assignees must
review and evaluation the information provided in the bug report on how to recreate the
problem. They must evaluate any technical information included inline, such as stack
traces or error messages, or as attachments, such as screen shots. The list of codes in
Appendix A reveals the wide variety of actions and information in use in bug reports.
Instances of negotiation are very common during this phase of the SWPM process. Negotiation turns may be taken by the assignee, the QAContact, commentators, and sometimes the reporter. Issues commonly negotiated during this phase are “whose responsibility is it?” if evidence suggests that the cause of the problem has not be determined with certainty; “what is the scope of the bug?” if there is disagreement or uncertainty about a reasonable community response to a complex problem, especially when a range of responses is possible. The issue of whether the bug report is a duplicate or not is not always resolved during triage, but continues past the original triage period. In some cases, a bug report is identified as a duplicate after it has been active for several weeks (see BR-C in Figure 5-1). Negotiation about whether the phenomenon reported is actually a problem or not may occur, sometimes centering on whether the behavior is a problem, an effect of the intent of the design, or it represents a request for new functionality (an enhancement). Negotiation occurs regarding whether the cause has been determined; how the bug report should be managed (e.g., complex problems are sometimes divided up and the pieces of the problem represented by separate bug reports). Issues of fix scheduling, priority of fixing a problem may be subject to negotiation. Negotiation about whether the changes made to the system comprise a fix is a commonly negotiated issue during this phase. Finally, the most common issue negotiated during this phase concerns the best design for the fix.

**Phase Four: Verification and Closure**

As mentioned earlier, bug reports are rarely set to “closed” in this community’s bug report repository. In most cases, bug reports are considered finished when their status is set to “verified.” The general process for determining whether a bug report can be set to this terminal state involves the bug report’s QAContact re-running the steps originally provided by the reporter in an attempt to cause the problem to happen again. This verification test occurs after the assignee changes the bug report’s status field to “resolved.” In the canonical case, the implication is that the assignee made a change to
the code that she or he believes addresses the phenomenon encountered and documented by the reporter. If the system behaves as expected, the QAContact will set the bug report’s status to “verified.” If the system does not behave as expected, or the QAContact has a different interpretation of what the correct system behavior should be, the QAContact can set the bug report status to “re-opened,” which returns responsibility for the bug report to the assignee. The key information practice in this phase is the evaluation of the system behavior compared to the problematic behavior described by the reporter. The most common issue negotiated at this juncture is whether the problem is fixed or not.

Verification takes place under other circumstances, too. Bug reports can be resolved by being fixed (described above) or by being identified as duplicates, as invalid or “won’t fix.” According to protocol, bug reports should be marked verified no matter what their resolution. Thus it is common to see QAContacts verifying the resolution of bug reports by virtue of their identification as duplicates, etc.

Another fairly frequent occurrence is bug reports never being verified; instead, they remain in some active state or in some valid resolution state forever.

**Discussion**

Table 7-1 presented a summary of how the SWPM process enacted by this F/OSS development community differs from the normative model and the ways in which the current project enriches our understanding of the organizationally situated, collective process of SWPM.

As noted earlier, the F/OSS community uses what is in many respects the same SWPM process used for many years by traditional, bureaucratic software development and systems management organizations. Some of the differences may be attributable to the differences in the organizational contexts. The F/OSS community is flatter and less bureaucratic than most traditional software development organizations although recent research suggests the F/OSS projects still maintain de-facto hierarchical structures (Sack
et al., 2004). There appears to be less coercion done in terms of motivating participants to manage software problems in the F/OSS community than would be common in a traditional environment. The existence of a large number of open bug reports (bug reports whose status is not yet “verified” or “closed”) that seem to be forgotten or neglected (because they have not been changed for many months) suggests that open bug reports are not reviewed regularly, which is a common practice in most traditional software development organizations. This project does not address the question of why so many bug reports are neglected, however.

One important finding from this project is the evidence the extent to which software problem management (SWPM) is primarily the management of information objects, bug reports. Management of bug reports is primary because 100-200 bug reports are created every day. Thus, the triage process must be efficient to prevent the community from being overwhelmed by new bug reports. We can recast the dictionary definition of triage in terms of bug reports: “the sorting of and allocation of resources to bug reports … according to a system of priorities designed to maximize the use of resources and minimize disruption to system users.” Each new bug report is evaluated in a comprehensive, holistic manner when it is first presented to the community / bug report repository before any attempt is made to resolve the problem, if a problem is deemed to exist.

The F/OSS community studied here is largely a community of volunteer developers and testers, with a smaller core of dedicated members. Given this community’s high rate of bug report ingest, one of the community’s strategies is quick elimination of as many of the new bug reports as possible, by marking them resolved as duplicates of other bug reports, as invalid (not real problems), won’t be fixed (either not a real problem, or, while perhaps a good thing to do, is beyond the scope of the current system; for example, rejected requests for enhancement).

The triage process can be interpreted as a sensemaking strategy if you consider the creation of a bug report as a signal that the normal state of affairs (working, healthy software) has broken down. Weick (1995) writes “to understand sensemaking is also to
understand how people cope with interruptions” (p.5). The receipt of an e-mail notifying someone of a new bug report is an interruption. The purpose of a SWPM process is to maintain or improve the health of a software system. Two of Weick’s seven properties of sensemaking also serve to explicate bug report triage in terms of sensemaking. The first is the notion of extracted cues (pp. 49-55) and the second is the notion of plausibility rather than accuracy (pp. 55-61).

In this community’s SWPM process, extraction of cues from bug reports is critical to the effective filtering and classifying of new bug reports. The reduction of the size of the stream of new bug reports by filtering out duplicate, invalid and “won’t fix” bug reports helps the community prioritize the work remaining for those responsible for taking a bug report through the “expert” phase (phase three). The community members involved in triage are also interpreting and authoring by evaluating reporters’ (1) claims of priority and severity and (2) their initial assignment of responsibility for the bug report through their selection of product, component, etc., when they create the bug report. The triage process consists of noticing that a new bug report exists and then, in some combination or sequence which the texts of the bug reports cannot completely reveal to us, continuing the sensemaking process by evaluating the evidence presented by the reporter, interpreting that evidence, making changes to the bug report (e.g., modifying priority; changing assignee; asserting that the current bug report is a member of a bug report network), adding interpretation (e.g., adding a comment), dynamically modifying the cooperative ensemble associated with the bug report (e.g., by adding those community members to the CC: list or changing the assignee). The changes made by the people performing triage constitute authorship of information for the use by the rest of the community.

In terms of “plausibility rather than accuracy,” the primary objective at this point in the process is moving the SWPM process forward relative to the current bug report based upon plausible information, interpretation, evaluation, and planning, not necessarily accurate information, interpretation, evaluation, and planning. Taking steps during triage sets the community in motion to take a next step; taking the next step or next action
results in the generation of additional information that is added to the bug report. That information can show (perhaps implicitly) that the course set during triage was accurate or it may reveal that a course correction is required. “It is more crucial to get some interpretation to start with than to postpone action until ‘the’ interpretation surfaces (Weick, 1995, p.57).” Triage example 7-2 above is an example of how quickly (in about one hour) the community is able to define a provisional plausible cause, gather and interpret additional information, adjust the social order by adding people to the dynamic cooperative ensemble, and then adjust and refine the “plausible” cause.

Designing a process where different people play distinct roles is a common and effective information and process management strategy. The result is that actions considered and taken by the community are done in an open and visible way, and “arms-length” transactions can be used at appropriate points in the process. Support for this kind of openness in an organization is known as due process, which can be defined as evaluating and reconciling, according to some established norms of behavior, evidence presented for evaluation from different, possibly contradictory, points of view (Gerson & Star, 1986). The notion of due process is particularly apt when considered in a context where multiple alternative actions are possible, optimal decisions are difficult to identify, and decision-making often occurs in the open, with opportunities for disagreement and the presentation of new evidence from different members of the community. Division of responsibility can also support the development of specialists, like triage or QA specialists in this community.

In this community’s SWPM process, the separation of duty between assignee and QAContact is one point where this kind of arms-length, open process is enforced. Another is point in the process is code review and code super review, where the assignee has his or her proposed software change reviewed by multiple independent experts. Code reviews are a common and widely acknowledged method for improving software quality, but the role of arms-length relationships and due process in ensuring quality has received far less attention (for an exception, see Sandusky, 2003).
Sensemaking theory (Weick, 1995) provides a powerful explanatory lens with which the nature and motivations of SWPM work can be examined. Based upon the analysis of the bug reports done in this project, and combined with my own experience using similar bug report repositories in several software development and systems management organizations, sensemaking is an appropriate way to explain the process by which software problems are managed. Sensemaking theory cannot on its own provide us with insights and understanding about the relationships between information, activity, process, and social order in distributed work. When combined with empirical investigation, like that conducted as part of this project, sensemaking can provide a unifying interpretive thread to help us understand the SWPM process.

As Weick notes, sensemaking is more than interpretation (i.e., the reading of an existing text): sensemaking addresses “how the text is constructed as well as how it is read. Sensemaking is about authoring as well as reading” (Weick, 1995) (p.7). In SWPM work, the bug report as created rarely contains information sufficient for its resolution. As the examples of triage (above) and negotiation (below) show, community members read what is there, take additional action, report the results of their action, suggest plans and alternative courses of action, and bring others into the management of the problem in an effort to make sense of the evidence, the symptoms, the phenomena described and experienced by the reporter. The reporter begins the process by authoring the description; subsequent contributors iteratively read and interpret the existing text, but also add to it in turn.

**Negotiation**

Negotiation is not a step or a phase in either the normative or the revised SWPM model discussed above. Negotiation is instead a basic social process that is employed as a means, in Strauss’ terms, to “get things done.” Negotiation frequently occurs during phases two, three and four (triage, expert analysis, and verification) of the SWPM process (see Table 7-1).
There are multiple contextual levels present in SWPM within the F/OSS community as described in Chapter Six. At the higher, structural level (Strauss, 1978), are issues that are infrequently or never the subject of negotiations in the bug reports. Examples of these are the format of the bug report, the bug report repository, or the SWPM protocol in use, which comprise the coordination mechanism (Schmidt & Simone, 1996). For example, the normative SWPM process (Table 7-1), based upon practice in traditional software development organizations, is not questioned or subjected to negotiation in individual bug reports: the SWPM process is an accepted part of the context of managing each bug report. The particular values available in controlled vocabulary fields within the bug report, like status and resolution, are not commonly negotiated; on the other hand, the names of the products and components are more likely to be changed in order to keep pace with the changing structure of the software itself, and thus could be negotiated issues (although the sample reported on here provides no evidence that issues such as these are negotiated within the bug reports themselves).

SWPM in any organization, whether traditional or F/OSS, is a negotiated order: it could always be otherwise. As Crowston (1997) points out, the coordination mechanism used to support similar processes can vary from organization to organization. Or, within an organization, different choices can be made, for example, about how programmers are assigned to particular bug reports. The focus of this chapter is not on negotiation in the large, or at the structural level, but on negotiation in the small, as it pertains to the management of individual bug reports.

The nature of this F/OSS development community is another aspect of the structural context surrounding SWPM and negotiations therein. For example, F/OSS communities are often characterized as meritocracies (Raymond, 1999) in contrast to traditional, bureaucratic software development organizations. While examination of these factors is beyond the scope of this project, it is useful to acknowledge that factors seemingly far removed from the management of a particular bug report might play a role in shaping the community’s actions and the outcomes.
The focus of community members directly involved in SWPM is on the resolution of particular software problems, and even more accurately, on the disposition of individual bug reports (remember that bug reports do not always represent software problems). The community implicitly distinguishes between what is and is not negotiable in order to successfully ingest and process this volume of bug reports. The coordination mechanism – the bug report and the associated protocol – are considered fixed parts of the structural context. The issues represented by individual bug reports are considered to be fluid and subject to negotiation.

Negotiation occurs frequently as members of this F/OSS development community manage software problems. Table 7-4 shows that a majority (61%) of the bug reports in the sample contain instances of negotiation. 27% of the sampled bug reports contain negotiation on more than one issue.

Based upon the empirical investigation described here, instances of negotiation vary along several dimensions, similar to Strauss’ dimensions of negotiation context (discussed in Chapter Six):

- the issue negotiated (see Table 7-5)
- level of intensity (civil vs. confrontational)
- duration (measured either by the number of distinct contributions to the negotiation sequence or the elapsed time from the start to the end of the negotiation)
- number of participants (number of distinct contributors)
- roles of the participants
- number of potential outcomes (distinct points of view, etc.)
- effect (impacts of distinct outcomes)
- status (the disposition of the negotiation)

For example, the level of intensity of an instance of negotiation can be more or less civil, democratic, or even sometimes rude. The tone is typically civil, but in one bug
report (BR 40), one negotiator begins his comment with “What’s wrong with you people?” In terms of duration and number of participants, we have found examples in our sample of 12 contributions from 5 people and 17 contributions from 4 people.

<table>
<thead>
<tr>
<th>Distinct Issues Negotiated per BR</th>
<th>Percentage of BRs</th>
<th>Cumulative Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>5</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>4</td>
<td>4%</td>
<td>5%</td>
</tr>
<tr>
<td>3</td>
<td>10%</td>
<td>15%</td>
</tr>
<tr>
<td>2</td>
<td>12%</td>
<td>27%</td>
</tr>
<tr>
<td>1</td>
<td>34%</td>
<td>61%</td>
</tr>
<tr>
<td>0</td>
<td>39%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Community members negotiate about bug priority; bug resolution scheduling; code reviews; whether a bug report represents a bug, an intended feature or an enhancement; whether the bug’s cause has been determined; how a bug report should be managed by the community; whether the bug’s cause has been identified; whether the bug is fixed yet or not; whether a problem exists or not; what the scope of the bug is; what type of bug is it; who is responsible for fixing the bug.

While F/OSS community members engage in negotiation about many aspects of the SWPM process, negotiation of (1) whether or not a problem exists and (2) issues of system design (insofar as design relates to the management of software problems) are the most common. Bug reports are rich in proposals for designs that can be used to fix the bugs represented by the bug reports. The participants are almost always geographically distributed. Thus, design negotiations in support of SWPM are open, distributed, and collaborative in support of the continuous (re)-design of the F/OSS system.

The definition and selection of a correct design to fix a problem is the second most frequently negotiated issue, occurring in about one-third of the bug reports in this sample. These negotiations are usually focused on relatively bounded aspects of the system because they occur in the context of trying to solve a particular software problem.
(see example 7-3, below). However, some bug reports are explicit requests for enhancements (example 7-4), some of which may be larger in scale. Negotiation about design is often necessary because the ideas for fixing problems must be developed and expressed. There are at least two situations that call for explicit expression of design ideas: in some cases, there may be multiple ways to bring the behavior of the system back into alignment with expectations. In other cases, a community member, usually the person assigned to fix the bug or the person reporting the bug, wants to express the approach she or he believes best addresses the bug in order to allow other community members to comment on the proposal.

**Table 7-5 Issue Frequency in Sample**

<table>
<thead>
<tr>
<th>Issue Negotiated</th>
<th>Percentage of BRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there a problem?</td>
<td>25%</td>
</tr>
<tr>
<td>What is best design?</td>
<td>22%</td>
</tr>
<tr>
<td>How should the BR be managed?</td>
<td>14%</td>
</tr>
<tr>
<td>Is this a duplicate BR?</td>
<td>13%</td>
</tr>
<tr>
<td>Is this problem fixed?</td>
<td>8%</td>
</tr>
<tr>
<td>Who’s responsible for fixing this?</td>
<td>7%</td>
</tr>
<tr>
<td>When should fix be scheduled?</td>
<td>5%</td>
</tr>
<tr>
<td>What is this BR’s scope?</td>
<td>4%</td>
</tr>
<tr>
<td>What is this BR’s priority?</td>
<td>3%</td>
</tr>
<tr>
<td>Is the fix’s code acceptable?</td>
<td>2%</td>
</tr>
<tr>
<td>Bug, feature, or enhancement?</td>
<td>2%</td>
</tr>
<tr>
<td>Is bug’s cause determined?</td>
<td>1%</td>
</tr>
<tr>
<td>What type of BR is it?</td>
<td>1%</td>
</tr>
</tbody>
</table>

In some cases, a bug report contains a single statement related to a negotiable issue. No one else ever comments on the statement, whether to support it, propose an alternative or present an argument against it. We consider these as examples of *implicit* negotiation:

Some negotiations may be very brief, made without any verbal exchange or obvious gestural manifestation; nevertheless, the parties may be perfectly aware of “what they
are doing” – they may not call this negotiating bargaining, but they surely regard its product as some sort of worked-out agreement. Other negotiations may be so implicit that the respective parties may not be thoroughly aware that they have engaged in or completed a negotiated transaction. If the latter kind of agreement gets broken by one person, however, the other is sure to experience some feeling, whether surprise, disappointment, annoyance, anger, or even a sense of betrayal or exploitation, but possible also relief or unexpected pleasure (Strauss, 1978) (p. 224-225).

In some of these cases, the bug report is subsequently marked as a duplicate of another bug report and thus no further comments are likely to be added to that bug report. The same issue may also, however, be negotiated in the related bug report.

In the following examples, identities of individuals are changed and some system-related names are disguised (framed by angle brackets: thus Microsoft Windows becomes <Commercial Operating System>). The first example (example 7-3) shows a simple design negotiation that includes the rationale behind the two points of view expressed by the participants. Example 7-4 is drawn from a bug report that represents a request for an enhancement rather than something in the system that is broken. This example also shows the influence of the community’s values, in this case ease of system use by multiple user constituencies, on the design. The third example is more complex and includes evidence of concurrent negotiation on 4 different issues: What is best design?, Who’s responsible for fixing this?, How should the BR be managed?, and Is this a duplicate BR? The participants in the negotiation in this BR also participate in negotiations and activity in the other bug reports that are part of the same bug report network.

**Negotiation Example 7-3, Bug Report 200:** In bug report 200, the preferred way of handling of one aspect of the user interface – the collapsible toolbars – is the subject of a design negotiation. The bug, which appeared as a regression after a normal system build, was that application toolbars, after being collapsed (that is, made invisible) by the user, could no longer be made visible – the interface “tabs” or “handles” were no longer on the interface. The report states in a comment on day 1:
On day 2, commentator_1 asks:

Commentator_1: Why is it suggested that the correct fix would be to disallow collapsing of the "File Edit..." toolbar? When this toolbar behaved correctly, the collapsed toolbar were easily retrieved by clicking on the little tab. What happened to the little tabs? That's the problem. [Turn 2: What is best design?]

The reporter responds within one hour on day 2:

Reporter: In comparison to <application name>, the "File Edit ..." menu toolbar shouldn't be collapsible. Once can argue as long as we could provide a way to expand the collapsed toolbars then things would be fine. However, this may seem odd when comparing to other apps. [Turn 3: What is best design?]

As it turns out, the reporter's argument that other applications don't provide collapsible toolbars failed to convince the community and the “File Edit...” toolbar remains collapsible to this day. In Example 1, misalignment about generally proper application behavior occurs and is expressed publicly in the bug report. Two different points of view are expressed in a pointed but not hostile manner in this regression bug that was marked fixed on day 3.

Negotiation Example 7-4, Bug Report 196: Bug report 196 is a request for an enhancement, not a report of a bug. Someone asked for the ability to create versions of the system more selectively. That is, the enhancement requestor, commentator_2, wanted to be able to leave a major, but optional, component of the system out.

The reporter starts the design negotiation by saying at the time the bug report is created (day one):

Reporter: This may be terribly easy: i.e., simply conditionally excluding the <component> meta module in <systemName>/modules/staticmod if the DISABLE_<component> variable is set. [Turn 1: What is best design?]

The first commentator replies at 11:04 on day one:

Commentator_1: except that the installer is expecting <component> so we need to use even yet another manifest file for creating packages (i'm not sure if we have to
use another set of .jst files in xpinstall, or if the .xpi files are created solely from the directories that the manifest creates). [Turn 2: What is best design?]

The reporter responds at 11:45 on day one:

Reporter: Who cares about the installer? This is for people that are building the <systemName> tree in order to embed it. [Turn 3: What is best design?]

The module owner (commentator_1) replies at 16:18 (day one):

Commentator_1: maybe the *installer* you don't care about, but embedders still probably want a functional list of stuff to embed. Are all the embedding clients keeping their own list of what they pull out of dist? [Turn 4: What is best design?]

Assignee_1 creates a patch and adds this comment at 18:44 on day one:

Assignee_1: Created an attachment (id=44324) simple fix, just like <reporter> suggested [Turn 5: What is best design?]

The bug report is untouched for several days, until day eight when assignee_1 submits another, much more complex patch. The bug report doesn’t record anything between day one and day eight, but it’s possible that the people working on this problem and/or other community members were communicating using other means (chat, email, face-to-face, etc.).

At 12:13 on day nine, after the second patch is code-reviewed by the reporter, the original requester, commentator_2, writes:

Commentator_2: minor enhancement request....when <systemName>_DLL is set, <systemName>.exe should link to <subsystemName>.dll rather than statically linking the same libraries. That will make isolating bugs much easier by having a testbed <systemName>.exe loading the same <subsystemName>.dll as an embedded client. [Turn 6: What is best design?]

Later on day nine, at 12:20, assignee_1 submits a third patch:

Assignee_1: Created an attachment (id=45281) implement <commentator_2>’s suggestion (not really tested yet) [Turn 7: What is best design?]
On day ten, at 13:16, the reporter, who seems to have been code-reviewing assignee_1’s four patches and, to some extent, guiding her work (note her comment “just like <reporter> suggested”), assigns the bug report to himself. Comments continue to contain a high level of code detail as the ramifications and side effects of the enhancement emerge and are addressed. The reporter submits another seven patches before the problem is resolved on all supported platforms (Windows, Macintosh and Unix/Linux). Commentator_1 takes on the role of code reviewer for the reporter’s patches. The bug report is closed on day 29, after 35 comments and 11 patches are posted and two formal code reviews are conducted. Implementing this enhancement did not turn out to be “terribly easy.”

Most of the design negotiation took place early in this bug report’s trajectory and was concerned with the overall ramifications of making this change on different kinds of system users and situations of usage. Four individuals participated in the negotiation making 7 contributions during the first 9 days the bug report was open. The reporter clearly was advocating for making the change while commentator_1 presented challenges or objections. The final contribution to the design negotiation was a request for a further enhancement, made by commentator_2 between assignee_1’s second and third patches. This seems to have been accepted without any response, as assignee_1’s third patch incorporates commentator_2’s request.

Design negotiation occurred here, but it seems to have failed in some senses. The design negotiation failed to reveal the scope of the effort either in lines of code, elapsed time or degree of effort. It did, however, bring the enhancement into the open and make it available for comment, which occurred in an open and constructive way (Gerson & Star, 1986). The bug report also provided a common space where the requester (commentator_2), the advocate / implementer (reporter), the first assignee (assignee_1) and the skeptic (commentator_1) could negotiate both the scope and merits of the request as well as manage the work of implementing it. The assignees and code reviewers could report status. Other commentators could make suggestions and provide encouragement.
Turns 2 through 4 in example 7-4 show an example of the interrelationship between the design of the system and the effects of design decisions on different types of users. “Installers” are those who simply want to install this system and run it. “Embedders” are people who are interested in taking this system, or selected components of it, and include as a subsystem it in other systems. This example shows, through this instance of negotiation, that managing bug reports software changes is not purely a technical issue.

**Negotiation Example 7-5, Bug Report 147:** This bug report contains instances of negotiation on four separate issues. The instances of negotiation run concurrently; they all overlap during the life cycle of this bug report.

The reporter begins negotiation on three of the topics in the description area of the bug report (day 1):

Reporter: Behavior should be consistent and subject to control by event handlers... Expected Results: Forms with a single input are submitted, even if a keypress event handler attempts to prevent it with stopPropagation() and preventDefault(). [Issue 1, turn 1: What is best design?]

Reporter: I'm guessing at the correct component. <Component_1> or <Component_2> are other obvious possibilities, but this seemed like the best fit. [Issue 2, turn 1: Who's responsible for fixing this?]

Reporter: If it's decided that letting Enter submit is desirable, this is probably two bugs: one to get Enter behaving consistently, one to get the event handler's efforts respected. [Issue 3, turn 1: How should the BR be managed?]

On day 4, a commentator begins the fourth negotiation instance on the topic Is this a duplicate BR?

Commentator_1: See also bug <X>, "Enter in text input submits form iff there is exactly one text input". [Issue 4, turn 1: Is this a duplicate BR?]

The reporter adds to two of the instances of negotiation on day 5. This marks the termination of negotiation on the issue “What is best design?” in this bug report.

Reporter: If <commentator from bug X>‘s spec is implemented in such a way that Enter keypress events on the input in a single-input form can be caught, my concerns would be completely addressed. (I don't actually much care what the
default behavior is, as long as I can catch the event and override that behavior.)

[Issue 1, turn 2: What is best design?]

Reporter: Similar but not identical. <commentator from bug X>'s proposed spec would address at least part of my problem, since he proposes activating whatever submit control is appropriate given his rules, and my <form> has no submit control. Presumably, in the absence of such a control, the form would not be submitted.

[Issue 4, turn 2: Is this a duplicate BR?]

At this point, the bug report records no activity for almost two months. On day 61, a second commentator makes an implicit contribution to the negotiation on who’s responsible for fixing this by changing the component field to a new value (changing it from “event handling” to “user interface design”). Changing the component field automatically changes the assignee and QAContact fields, an implicit re-attribution of responsibility by re-assignment of the bug report.

Commentator_2: Moving to UI/design... [Issue 2, turn 2: Who’s responsible for fixing this?]

On day 63, the remaining three instances of negotiation are concluded. The issue of “Who’s responsible for fixing this?” is firmly addressed by the new assignee, who turns it back to the original assignee and concludes the negotiation on this issue with this comment and by resetting the component field to the value chosen by the reporter on day 1:

Assignee_2: This ain't a UI design problem, it's an Event Handling problem (or possibly an <Component_2> problem). [Issue 2, turn 3: Who’s responsible for fixing this?]

Later on day 63, the assignee concludes the remaining two negotiations in this bug report in a single, complex comment:

Assignee_1: The fact that you can’t cancel the submission with the enter key is a dupe of bug <Y>. A discussion of the second part about whether or not we should submit forms with multiple text fields on Enter is covered in bug <X>. *** This bug has been marked as a duplicate of <Y> *** [Issue 3, turn 2: How should the BR be managed?] and [Issue 4, turn 3: Is this a duplicate BR?]
Example 7-5 clearly illustrates the complex and contested relationships between a bug (the observed, experienced system phenomenon) and bug reports (the socially constructed representations of bugs and other types of things, like bug report networks, parties, etc.). In this case, the reporter did experience misalignment between the system’s behavior and expectations. However, this bug report didn’t get “fixed;” it got marked as a duplicate of bug report Y. In addition, as a result of the negotiation regarding how the bug report should be managed, the community decided that another portion of the reporter’s issue would be tracked by a third bug report (bug report X). The reporter added a comment reiterating the points he made in this bug report (see comments marked “issue 1, turn 1” and “issue 1, turn2,” above) to bug report X one day after this bug report was marked resolved: duplicate. The reporter also registered an interest in bug report Y by adding himself the “carbon copy” list to ensure he received e-mail updates on the progress on and changes to bug report Y. The reporter was still invested in seeing that the problem he had reported was actually resolved. He was also willing to abide by the community’s decision regarding how the bug report he had created should be managed (issue 3). Assignee_2 was active in the eventual resolution of bug report X, but did not play a specific role (e.g., assignee, QAContact). Assignee_2 was never recorded as a participant in bug report Y. Assignee_1, on the other hand, was very active throughout the history of bug report Y, acting as the initial assignee, yet never participated in bug report X.

Discussion

Negotiation is a basic social process commonly used in a variety of contexts by this community to support SWPM (Tables 7-4 and 7-5). 61% of the bug reports in our sample contain negotiation on at least one issue and 27% contain evidence of negotiations on more than one issue. Because our sample size is representative of the population (as discussed in section 2 above), an estimated 110,000 of the 182,000 bug reports in the repository contain instances of negotiation and nearly 50,000 of those
110,000 contain instances of negotiation on more than one issue. Negotiations can occur regarding whether a bug report represents an actual bug or a request for system behavior that is not, should not or will not be provided (“Is there a problem?”). Negotiations often arise as the community tries to understand whether the reported bug is actually fixed or not (“Is this problem fixed?”), which can relate back to differing expectations about what a correct, complete or sufficient fix is (“What is best design?”). There are several other issues that are subject to negotiation, including bug report management (“How should the bug report be managed?”), fix scheduling (“When should fix be scheduled?”), duplicate bug report identification (“Is this a duplicate bug report?”) and who is responsible for fixing a bug (“Who’s responsible for fixing this?”).

A bug, by its nature, represents misalignment within the community. A user has noticed some behavior in the system that is at odds with established standards or expectations. These expectations can be set formally by the community (e.g., conformance to a standard) or be the expectations of the user (e.g., when I select an interface element that looks like X, I expect behavior Y). In order to re-align system behavior with expectations, the community must come to agreement about (1) the actual desired behavior of the system and (2) what actions are necessary to effect re-alignment. In some bug reports, evidence of explicit negotiation doesn’t exist: in these cases, what needs to be done somehow seems obvious to the community members (e.g., this bug report is obviously a duplicate of another bug report). In other situations, the negotiations can be long and involve many community members.

This work informs and extends existing theories of coordination by examining the nature of SWPM at a different, finer grained level of analysis, the individual bug report. Coordination theory is primarily to be used to analyze processes and identify ways to improve them (Crowston, 1997). Crowston analyzes how dependencies between activities are managed in the software change process, focusing on a higher, process-level view. However, Crowston presents no information about how activity and information are used to manage dependencies arising within individual cases of software change.
Schmidt and Simone (1997) also focus on the process level by arguing that a combination of artifact and protocol together provide a coordination mechanism. The bug repository is the central coordination mechanism in this F/OSS development community. The repository supports SWPM by enacting an artifact (the bug report) and a protocol (standard, but flexible, sequences of activities). Schmidt and Simone’s theory does not, however, predict or explain why negotiation is so prevalent in the field of work, within individual bug reports.

In Strauss’ terms (1978), social order is negotiated order. Strauss uses the term structural context to describe “that ‘within which’ the negotiations take place, in the largest sense” (Strauss, 1978) (p. 98). What we can see and are analyzing here, however, is negotiation applied at a lower contextual level, within individual bug reports. Strauss uses the term negotiation context to describe the “properties entering very directly as conditions into the course of the negotiation itself” (p. 99). Bug reports, as dynamic artifacts, have different (and potentially unique) groups of people associated with them: the various reporters, assignees, commentators, and so on. Negotiation in bug reports represents negotiation of both information and social order within each of these sub-communities. Negotiation at the bug report level is emergent, dynamic, and organic in the service of managing both bug reports (which may or may not represent problems) and the software problems that some bug reports represent. In contrast, the bug repository – this community’s coordination mechanism – changes far more slowly and within the parameters of the normative SWPM process described in Table 7-1. Crowston’s and Schmidt and Simone’s theories can be applied to the analysis of structural context (e.g., coordination mechanisms), like a SWPM system, but they cannot be applied to the study of individual instances of negotiation.

SWPM is also a sensemaking process (Weick, 1995). Given problematic phenomena experienced by a user, tester, or developer, a bug report may be created to represent those phenomena. While some of the symptoms, conditions, and evidence may be known and entered into the bug report, other questions may remain about the causes of the phenomena; the boundaries of the suspect system behavior; where the issues raised
by a bug report fit within the constellation of the community’s values; what approach should be taken by the community in response to the bug report; what code changes, if any, should be generated to re-establish alignment. The analysis here shows negotiation occurring in a majority of the bug reports in our sample. Negotiation is one of the basic social processes employed by the community to move toward consensus regarding issues of behavior, values, design, causes, and code, and thus to problem resolution.

Instances of negotiation in bug reports make many aspects of SWPM visible. When, in a design negotiation, more than one point of view is expressed, it may be that no single point of view prevails, but that more than one approach contributes to the implemented changes. The design of a system is not only in what is expressly included, but also in what is excluded, in a positive ground / negative ground relationship. Design negotiations also give us information about the community’s values when participants include the rationale for their points of view, as in example 7-4 (above). Recovery of design rationale, especially in F/OSS development communities that produce little or no formal documentation, motivates continued investigation into the activities and information represented in bug reports.

Community Re-enactment

The processes by which new participants in this community’s SWPM process are introduced and come to learn the processes and norms of behavior are largely beyond the scope of this project. However, one pattern occurred frequently enough to warrant a brief description here.

Experienced community members sometimes use the triage process as an opportunity to improve the problem reporting skills of the bug report’s creator or to reinforce good bug report writing habits. Triage is performed by relatively experienced members of the community and many bug reports are created by people with little sense of the way the SWPM process is organized. BR 133 contains this positive amendment to a comment:
(Nice testcase though :-})."

BR 125 contains this advice:

Also, please read the bug reporting guidelines at http://<url>
and provide information mentioned therein in your bug reports (build id and so on).

BR 197 contains this indirect advice about how to report a problem:

Reporter, without a way to reproduce or an attachment, the best I can do is dupe
this against a bug with the same description.

**Conclusion**

This chapter has enriched the normative SWPM model originally presented in Chapter One by including insights from the analysis of the information practices, activities, processes and social order revealed during this empirical investigation. SWPM is not adequately represented by a one-dimensional engineering process model with a highly deterministic work flow: instead, it is more accurately represented and conceptualized as a complex, situated, and contingent amalgam of nested processes of different types. SWPM is itself one component of an ever more complex engineering process, software development. This project has revealed that the engineering-oriented SWPM model can be decomposed into four distinct phases: Problem Identification and Reporting; Bug Report Triage; Expert Analysis, Fixing, Testing and Deployment; and Bug Report Verification and Closure. SWPM is simultaneously an engineering process and a sensemaking process. Collective and individual debugging are finer-grained sensemaking processes that are primarily part of phase three. Negotiation is a basic social process that occurs during the sensemaking processes: SWPM and collective debugging. Negotiation can also be a signal of a shift from individual debugging to collective debugging.
Phase one, problem identification and reporting, begins with the human encounter with problematic phenomena and concludes with the translation of that encounter into an information object, the bug report. The primary actor is known as the reporter, usually the person with direct experience of the problematic phenomena. Phase two, bug report triage, is primarily oriented toward management of the information object, the bug report, rather than management or correction of the problem. Phase three, expert analysis, fixing, testing and deployment, is primarily oriented toward correction of software problems. Much of the activity takes place outside of the bug report repository as code is changed and tested, but the bug report may be used as a site for other activities related to the management and resolution of the software problem (e.g., negotiation, questioning, hypothesizing, and reconfiguration of the social order). Much of the activity in this phase is debugging, both on an individual and collective level. The primary actor in this phase is the assignee, a software developer who has the knowledge and skill to design and implement, often with the participation of other community members, a fix for the software problem. Phase four, bug report verification and closure, occurs only after someone asserts that phase three has concluded and there is no more corrective action to be taken. The primary actor here is the QAContact whose goal is to confirm that the problematic phenomena has been eliminated and finally remove the bug report from the active field of work.

Negotiation has been identified as a common and important basic social process, employed by the actors involved in the management of particular bug reports during all phases of the SWPM process. One provisional finding is that the issues negotiated by community members vary depending upon which phase of the process the bug report is in. During triage, issues like whether a problem exists or not, whether the bug report is a duplicate of an existing bug report or not, who is responsible for fixing this problem are commonly subject to negotiation. During phase three, issues like what is the best design for a fix, how should the BR be managed, is this problem fixed, what is the BR’s scope, and who is responsible for fixing this are common issues for negotiation.
Negotiation, especially in light of how often it occurs, also reveals that SWPM, and software development as a whole, are contingent, situated, and contested practices, heavily influenced by contextual factors at many levels (see Chapter Six for discussion of the contextual layers relevant to SWPM).

Decomposing the larger SWPM process into phases, constituent sensemaking processes and basic social processes also helps reveal the complexity of the community’s use of information and employment of information practices in support of the community’s goal of maintaining or improving the quality of its software. Close empirical evaluation of the bug reports shows how and when particular information practices, like searching for duplicate bug reports, evaluating cues, or comparing symptoms, are employed. This decomposition also reveals opportunities for future research in varying directions. For example, a researcher could chose to build on this approach to develop an even more accurate model of SWPM in support of software engineering practice. Alternatively, a researcher could build on this approach to further develop models and theories of domain-independent information practices.

Other interesting processes remain to be examined in more detail. Some that were noted but not pursued in depth are voting; individual learning and the integration of individuals into the SWPM sub-community as a function of the SWPM process and the knowledge contained in bug reports. One aspect of learning and community re-enactment was noted above: the ways in which experienced community members try to improve the performance of problem reporters by providing feedback regarding their reports.
CHAPTER EIGHT: SOCIAL ORDER AND DISTRIBUTED WORK

Introduction

The manner in which the human actors in a distributed work community are organized, whether by hierarchical strictures or through dynamic reconfiguration, is as important an influence on the organization as the kinds of information, activities, and processes employed by the community. The degree to which social order is fixed or fluid is influenced by contextual factors as discussed in Chapter Six. Software problem management (SWPM) is a sensemaking process and the transformation of the situational and uncertain conditions presented by particular bug reports into solved software problems depends upon the participation, expertise, and knowledge of more than one individual. The reporter presents some evidence about the problematic phenomenon when he creates a new bug report. What happens next to the bug report is contingent, depending upon the evidence, the person triaging the bug report, the assignee, commentators, their negotiations, actions, and decisions, and so on. SWPM is a collective community response to the reporter and the phenomenon he has reported.

Previous chapters have discussed the formal roles directly supported by the bug report repository and associated with bug reports such as reporter, assignee, and QAContact. This chapter takes a closer look at these roles and also defines several informal roles that people play that have been identified during the empirical analysis of the bug report sample. The bug report network analyzed previously (see Chapter Five) is used here as a basis for discussing the dynamic formation of distributed collective ensembles, the sub-communities formed to manage each bug report.

Social Order in Distributed Work

Social order, as defined in the context of negotiation, is
The larger lineaments of groups, organizations, nations, societies, and international orders that yield the structural conditions under which negotiations of particular kinds are or are not initiated by or forced on actors. (Strauss, 1978) (p.12)

Another definition, using the words “social world” instead of “social order” reads:

“A social world is an interactive unit that arises when a number of individuals strive to act in some collective way, often requiring the coordination of separate perspectives and the sharing of resources. It has ‘at least one primary activity (along with related activities, … sites where activities occur … [and] technology (inherited or innovative means of carrying out the social world’s activities’ (Strauss, 1978; cited in Clarke, 1991, p.131).” (Fitzpatrick, Tolone, and Kaplan, 1995) (p.4) (emphases in original).

Social order, in any given context, is more or less stable. The social orders of national governments, modern corporations, and bureaucratic institutions may appear to be very stable, even if continuous incremental change occurs at the margins. Other social orders, such as alliances between political activist groups, may in contrast appear less stable.

The notion of social order can be applied at different levels of analysis when considering SWPM in this F/OSS community. The community as a whole forms a social order that may appear to be relatively stable. As is the case with many large-scale F/OSS projects, there is a core of people in the project who seemingly participate “full-time.” In F/OSS projects that combine paid workers from a corporation (e.g., Netscape employees working on the Mozilla project; IBM employees working on the Apache project) with the “volunteer” hackers commonly associated with F/OSS projects, the paid workers might also be considered part of a stable, core social order.

At the level of groups working together to get things done, social order may be relatively fluid. During any short-term software development project, people may be added to, removed from, or re-assigned to different roles on the project. In SWPM, in the context of individual bug reports and bug report networks, social order may be even
less stable. For example, evidence from the examination of problem management activity revealed in bug reports shows that reassignment of the bug report to different workers is common; people without a specific identifiable role make contributions in the form of comments, etc. (Determining the precise boundaries between core and non-core participants in the project under review is beyond the scope of the current project. This chapter focuses instead on the groups that form around each individual bug report.)

The term cooperative ensemble is used in the Computer-Supported Cooperative Work (CSCW) literature to denote the people involved in a cooperative work situation (Schmidt, 1991; Schmidt & Simone, 1996; Fitzpatrick, Tolone & Kaplan, 1995). While it is difficult to find a succinct definition of cooperative ensemble in the literature, Schmidt and Rodden (1996) state three characteristics of cooperative ensembles:

- Cooperative ensembles are either large, or they are embedded within larger ensembles
- Cooperative ensembles are often transient formations, emerging to handle a particular situation after which they dissolve again.
- Membership of cooperative ensembles is not stable and often even non-terminable. Cooperative ensembles typically intersect.

There is scant research that is focused on the nature, formation, and evolution of cooperative ensembles. As other researchers have pointed out, people involved in cooperative work are not necessarily always in agreement or working in harmony. As Schmidt states, cooperative ensembles are coalitions of people whose work is “distributed, not only in terms of time and space but also, and more importantly, in terms of ‘control,’ i.e., in terms of contingencies, individual heuristics and biases, incongruent specialisms and incomplete perspectives, divergent or conflicting motives and interests” (Schmidt, 2002) (emphasis in original, pp. 18-19).

Egger and Wagner's work on schedule negotiation amongst surgical team members (1992) implicitly addresses issues of the formation of cooperative ensembles.
Within a surgical clinic, the set of surgeons, anesthesiologists, nurses, etc., is limited. For each scheduled surgery, a set of people with appropriate skills and availability must be coordinated and assembled. This is an example of dynamically constructed, collocated cooperative ensembles.

In the case of the bug reports managed within the F/OSS development community studied here, cooperative ensembles occur at multiple contextual levels. The entire F/OSS development community (including users) can be said to comprise a cooperative ensemble in the large. In particular, those people who have been engaged in the use of the community’s bug report repository comprise a cooperative ensemble of particular interest. The membership of this cooperative ensemble is fluid as people move into and out of participation in SWPM; some members are more or less “permanent,” others “transient,” particularly one-shot problem reporters.

Each bug report has its own cooperative ensemble consisting, at a minimum, of the reporter, the assignee, and the QAContact. As noted above, bug reports also allow community members to register an ongoing interest in the bug report by attaching their e-mail address to the bug report’s CC: list. Other community members may participate as commentators without registering a persistent interest in the particular bug report. In many cases, but not always, a single software problem corresponds to a single bug report (see discussion in Chapter Five, above).

The third and final level of cooperative ensemble is associated with a bug report network (BRN). BRNs are created when community members assert formal and informal relationships between individual bug reports. Each constituent bug report has its own associated cooperative ensemble. When bug reports are related to each other, the individual cooperative ensembles associated with each distinct bug report are reconstituted into a (potentially) larger cooperative ensemble. In many cases, but not always, a single software problem corresponds to a single bug report network (see discussion in Chapter Five, above).

When a bug report is created, the creator’s identity is attached to the bug report in the role of “reporter.” When the reporter selects product and component values, the
bug report repository makes default assignments for assignee and QAContact based upon the definitions held within the repository and associated program logic. At the time of bug report creation, the assembly of the initial cooperative ensemble is *dynamic* and automatic. If and when other commentators participate or others are added as members of the bug report’s CC: list, the cooperative ensemble grows dynamically, but not automatically. In contrast to Egger and Wagner’s (1992) surgical clinic example, however, the members of any cooperative ensemble here are not collocated; instead, in a F/OSS development community, the assumption is that the members of the cooperative ensemble are *distributed*. Thus, the creation of cooperative ensembles at the bug report level is both dynamic (determined in real time as the trajectory of the bug report unfolds) and distributed (in terms of the physical location of the ensemble members). The *control* of the configuration of the ensemble is distributed, too: any registered member of the SWPM community is able to contribute comments to any bug report and / or attach his or her identity as a member of the bug report’s CC: list. Control over the individuals playing certain roles, like default assignee and default QAContact, is more closely restricted to those with particular expertise. The term *distributed cooperative ensemble (DCE)* describes this dynamic form of social order associated with the primary information object and unit of work in this domain, the bug report.

**Example 8-1: Distributed Cooperative Ensemble:** This discussion refers to the BRN described in Chapter Five and represented in Figure 5-1. First, we’ll look at how one distributed cooperative ensemble (DCE) forms around one bug report, BR-E. This bug report, modified to obscure the identity of the community and the participants, is presented in Figure 8-1, below.
Assigned To: <PC>   Reported By: <TB>   QA Contact: <null>
Component: General   Target Milestone: ---
URL:    Summary: <System> hangs at start-up
Keywords: regression, smoketest
Status Whiteboard:       Opened: <day 129> 02:49

Description:
<3 lines removed; RS>
System build <day 129> hangs at start-up, the splash comes up and nothing more happens and the application must be force quit.

Reproducible: Always
Steps to Reproduce:
Double-click on System icon

Actual Results:
Only the splash shows.

Expected Results:
Start up.

-------- Additional Comment #1 From <TB> <day 129> 02:50 --------
Forgot to mention that build <day 128> works fine.

Keywords changed from NULL to regression

-------- Additional Comment #2 From <MP> <day 129> 06:04 --------
i'm pretty sure this is caused by <BR-B>

-------- Additional Comment #3 From <MP> <day 129> 06:16 --------
i'm this is caused by <BR-B> -> <PC>.
Add MP to CC list
AssignedTo changed from <MP> to <PC>
Severity changed from critical to blocker
Keywords changed from regression to regression, smoketest

-------- Activity Log Entry By <TL> <day 129> 06:32 --------
Add <DM> to CC list

-------- Activity Log Entry By <TL> <day 129> 06:33 --------
Add <HT1> to CC list

-------- Additional Comment #4 From <MP> <day 129> 06:37 --------
Created an attachment (id=130059)
sampler report of hang
here's a report on the hang from sampler.

-------- Additional Comment #5 From <PC> <day 129> 06:48 --------
<MP> did you back out bug <BR-B>?

-------- Additional Comment #6 From <MP> <day 129> 06:48 --------
only locally, i did not check it in.

Figure 8-1 Text of BR-E
Figure 8-1, cont.

The DCE is formed when the bug report is created at 02:49 on day 129\textsuperscript{12}, with two members: the reporter, TB, and the assignee, MP. (In this community, DCEs normally contain 3 members when a bug report is created, but in this case, the QAContact role is not assigned, presumably because the repository has no default QAContact for this product / component pair. On day 130, the reporter, TB, takes the QAContact role by verifying the resolution.) The bug report is marked as representing a regression, a problem that was introduced by a recent change to the software. This bug report is opened at the second highest priority level, critical. The majority of the activity on this bug report occurs between 06:04 and 06:59 on day 129: the DCE expands to 6 members during this 55 minute time period. The problem is reassigned to PC (member 3) by the first assignee, MP (MP adds himself to the CC: list to ensure he continues to receive e-mail updates about this bug report). The severity is changed to the highest level, blocker. A commentator, TL (member 4), enters no text but adds two others to the CC: list (members 5 and 6) for reasons that are not specified. Finally, on day 130, after the resolution is completed and verified, the final member of the DCE, OC (member 7),

\textsuperscript{12} In this example, the day count begins at “1” with the creation of the earliest bug report in the bug report network, BR-A.
becomes involved by marking another BR (BR-F), as a duplicate of BR-E, and adds himself to the CC: list on BR-E. BR-E is identified, opened, and resolved in an extremely short period of time, in about 4 hours and 15 minutes.

The design of the repository – the coordination mechanism – plays a key role in shaping the response of the community members to this new bug report. The automatic e-mail notifications sent to the assignee, MP, trigger the rapid response. Many of the people associated with this BR are central members of the SWPM community here because their names occur frequently in the bug reports. They are experienced, and understand how to leverage the capabilities of the coordination mechanism, both the repository and the conventions and protocols traditionally used by the community, in support of efficient SWPM. MP, for example, knows to add himself to the CC: list when re-assigning the bug report to PC to ensure he continues to receive e-mail updates.

The defined formal roles reporter, assignee, and QAContact are insufficient for understanding the range of roles played by community members. A number of additional roles have been defined based upon the analysis of the bug reports in the sample. Table 8-1 summarizes, categorizes, and defines the range of formal and informal roles found in this SWPM community during the current project. The bug report repository (part of the community’s technical context) formally specifies three of these roles: the five additional roles are informal. BR-E and the bug report network within which it is embedded illustrate many of the formal and informal roles listed in Table 8-1.

Table 8-1 makes a first distinction between contributors and non-contributors. Contributors are those who participate in the management of a bug report by making any change that is recorded in the bug report itself, including creating the bug report, adding comments, modifying fixed fields, etc. Non-contributors are those who participate in a relative passive way by merely being aware of the bug report, either by inclusion in the CC: list or by reading the bug report but not being associated with the bug report in any traceable way.
Table 8-1 Formal and Informal Roles in Distributed Cooperative Ensembles (DCEs)

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contributor</td>
<td>Community members who do one or more of (1) reporting a problem (2) adding a comment (3) taking an action that appears on the activity log.</td>
</tr>
<tr>
<td>Reporter</td>
<td>The community member who creates the bug report; a formal role.</td>
</tr>
<tr>
<td>Assignee</td>
<td>Community member who is in the role of assignee for the bug report; a formal role.</td>
</tr>
<tr>
<td>QAContact</td>
<td>Community member who has responsibility for verifying and closing bug reports; a formal role.</td>
</tr>
<tr>
<td>Commentator</td>
<td>Community member who (1) adds a text comment, (2) creates an attachment, or (3) asserts relationships (including duplicate, dependency, and informal relationships) between bug reports.</td>
</tr>
<tr>
<td>Articulator</td>
<td>Community member who makes a change reflected in the activity log, but not a text comment.</td>
</tr>
<tr>
<td>Facilitator</td>
<td>Community member who adds names (including self) to the CC: list but otherwise makes no contribution</td>
</tr>
<tr>
<td>Non-Contributor</td>
<td>Community members who never comment or make changes to a bug report.</td>
</tr>
<tr>
<td>Observer</td>
<td>Member of the CC: list who never contributes to the bug report.</td>
</tr>
<tr>
<td>Reader</td>
<td>Community member who retrieves, looks at, or reads the bug report (there is no way to identify readers from the available data). I, as a researcher, am a reader.</td>
</tr>
</tbody>
</table>

Table 8-2 summarizes the BR-E’s DCE. Note how three of the members of the DCE associated with BR-E (TL, DM, and HT1) are associated as interested, informed observers, but not as active participants in the resolution of the problem. None of them ever enters a comment, but all of them are observing the activity around this bug report. It’s not possible to tell how TL, who adds DM and HT1 to the CC: list, knows so quickly about this BR. TL is neither on the CC: list nor registered as playing a particular role on this bug report. It is possible that TL is linked by one of the values set in a fixed bug report field (product or severity, for example).
TL is a facilitator, who adds two actors to the CC: list but makes no other contribution to the bug report. TB (reporter and implicit QAContact), MP (assignee_1), PC (assignee_2), OC (commentator) are contributors to the bug report. DM and HT1 are observers added to the CC: list by the facilitator TL.

**Table 8-2 DCE for BR-E**

<table>
<thead>
<tr>
<th>Contributor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reporter</td>
<td>TB</td>
</tr>
<tr>
<td>Assignee</td>
<td>MP,PC</td>
</tr>
<tr>
<td>QAContact</td>
<td>-- (TB, implicit)</td>
</tr>
<tr>
<td>Commentator</td>
<td>OC</td>
</tr>
<tr>
<td>Articulator</td>
<td>--</td>
</tr>
<tr>
<td>Facilitator</td>
<td>TL</td>
</tr>
</tbody>
</table>

**Non-Contributors**

<table>
<thead>
<tr>
<th>Role</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
<td>DM, HT1</td>
</tr>
<tr>
<td>Reader</td>
<td>unknown</td>
</tr>
</tbody>
</table>

It is not possible to draw any conclusions about the motivations, expertise, or potential for contribution that facilitators or observers may have relative to a bug report. Contributors leave some trace of their activity in the bug report. It is likewise impossible to know if there have been any readers of a bug report.

**Table 8-3 DCE for BR-A (Meta bug report)**

<table>
<thead>
<tr>
<th>Contributor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reporter</td>
<td>GK</td>
</tr>
<tr>
<td>Assignee</td>
<td>NA</td>
</tr>
<tr>
<td>QAContact</td>
<td>CH</td>
</tr>
<tr>
<td>Commentator</td>
<td>--</td>
</tr>
<tr>
<td>Articulator</td>
<td>--</td>
</tr>
<tr>
<td>Facilitator</td>
<td>MO</td>
</tr>
</tbody>
</table>

**Non-Contributors**

<table>
<thead>
<tr>
<th>Role</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
<td>--</td>
</tr>
<tr>
<td>Reader</td>
<td>unknown</td>
</tr>
</tbody>
</table>
### Table 8-4 DCE for BR-B

<table>
<thead>
<tr>
<th>Contributors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reporter</td>
<td>AR</td>
</tr>
<tr>
<td>Assignee</td>
<td>HT, PC</td>
</tr>
<tr>
<td>QAContact</td>
<td>AB</td>
</tr>
<tr>
<td>Commentator</td>
<td>AS, GK, WE, BE, TL, DBR, DM, BMA, JS, MP, HP, BZ</td>
</tr>
<tr>
<td>Articulator</td>
<td>DT</td>
</tr>
<tr>
<td>Facilitator</td>
<td>BME, AD, DK</td>
</tr>
<tr>
<td><strong>Non-Contributors</strong></td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td>NG, HD, DBA</td>
</tr>
<tr>
<td>Reader</td>
<td>unknown</td>
</tr>
</tbody>
</table>

### Table 8-5 DCE for BR-C

<table>
<thead>
<tr>
<th>Contributors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reporter</td>
<td>NG</td>
</tr>
<tr>
<td>Assignee</td>
<td>HT1, PC</td>
</tr>
<tr>
<td>QAContact</td>
<td>AB</td>
</tr>
<tr>
<td>Commentator</td>
<td>GK, AS</td>
</tr>
<tr>
<td>Articulator</td>
<td>--</td>
</tr>
<tr>
<td>Facilitator</td>
<td>--</td>
</tr>
<tr>
<td><strong>Non-Contributors</strong></td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td>--</td>
</tr>
<tr>
<td>Reader</td>
<td>unknown</td>
</tr>
</tbody>
</table>

### Table 8-6 DCE for BR-D

<table>
<thead>
<tr>
<th>Contributors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reporter</td>
<td>NG</td>
</tr>
<tr>
<td>Assignee</td>
<td>HT1</td>
</tr>
<tr>
<td>QAContact</td>
<td>AB</td>
</tr>
<tr>
<td>Commentator</td>
<td>GK</td>
</tr>
<tr>
<td>Articulator</td>
<td>--</td>
</tr>
<tr>
<td>Facilitator</td>
<td>--</td>
</tr>
<tr>
<td><strong>Non-Contributors</strong></td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td>--</td>
</tr>
<tr>
<td>Reader</td>
<td>unknown</td>
</tr>
</tbody>
</table>
Table 8-7 DCE for BR-F

<table>
<thead>
<tr>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reporter</td>
</tr>
<tr>
<td>Assignee</td>
</tr>
<tr>
<td>QAContact</td>
</tr>
<tr>
<td>Commentator</td>
</tr>
<tr>
<td>Articulator</td>
</tr>
<tr>
<td>Facilitator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
</tr>
<tr>
<td>Reader</td>
</tr>
</tbody>
</table>

Table 8-8 combines the all of the distinct identities present in each role over all six of the bug reports. The commas separate the actor identities associated with particular bug reports. There are exactly six reporters listed, one per bug report in the bug report network; note that two of the bug reports were reported by the same individual. In three of the bug reports, two different actors were assignees: their identities are separated by a slash (for example the first assignee for BR-B was HT and the second assignee was PC). Note the commonality in assignees across the six bug reports: there are five distinct individuals assigned to these bug reports at various times. Note also the low level of variance in QAContact. Consistency in the identity of assignee and QAContact is to be expected because these values correlate with the settings of the product and component fixed fields and because these six bug reports share significant characteristics (which is implied because of they are linked into a bug report network).

One surprising observation is how frequently the roles of facilitator (someone who adds someone to the bug report’s CC: list but makes no other contribution) and observer (someone on the CC: list who makes no contribution to the bug report). It’s not possible to say whether this pattern is common or not based upon data analysis to this point; quantitative examination of patterns of social order are beyond the scope of the current project. It may be that these bug reports, which represent a “critical” crash bugs,
cause a higher than normal level of interest among community members regarding the management of these bug reports and the problems they represent.

Table 8-8 DCE for the Bug Report Network Comprised of BR-A .. BR-F

<table>
<thead>
<tr>
<th>Contributors</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reporter</td>
<td>GK, AR, NG, NG, TB, OC</td>
</tr>
<tr>
<td>Assignee</td>
<td>NA, HT/PC, HT1/PC, HT1, MP/PC, HT1</td>
</tr>
<tr>
<td>QAContact</td>
<td>CH, AB, AB, AB, (TB), AB</td>
</tr>
<tr>
<td>Articulator</td>
<td>--, DT, --, --, --, --</td>
</tr>
<tr>
<td>Facilitator</td>
<td>MO, BME/AD/DK, --, --, TL, DK</td>
</tr>
</tbody>
</table>

Non-Contributors

| Observer           | --, NG, HD, DBA, --, --, DM/HT1, -- |
| Reader             | unknown |

The bug report data also lacks information about which actors play key roles in the community (e.g., module owners). This knowledge, if available, might help explain why particular identities appear in certain roles under certain conditions. It may be that the module owner is routinely added as an observer for many bugs associated with her module if only to ensure the opportunity to be heedful to the activities of the community vis a vis her area of responsibility.

Conclusion

The analysis above clearly shows that certain actions, when taken by community members, cause changes in the social order. For example, association of a new bug report with a product (or re-associating an existing bug report with a new product) causes an automatic change to the assignee (the person responsible for providing a fix for a problem) and the QAContact, dynamically establishing a particular social order at the bug report level. Community members also have the opportunity to make direct changes to fields like assignee and QAContact at any time. Assertion of relationships between
existing bug reports (creating or modifying existing bug report networks) also has an impact on social order. This effect is most obvious when a dependency relationship is created between two bug reports. In such a case two partially or completely disjoint DCE’s may merge, and this may increase the communication and coordination costs of managing the bug report(s).

The technical context, particularly the capabilities of the bug report repository, has a great deal of influence over when and how social order is established and how it evolves. The protocols, norms, and processes used by the community also influence how social order is established at the bug report and bug report network levels. Together, the repository and the associated processes form the central coordination mechanism used by this community to manage software problems.

The community members are facile in their use of the bug report repository as a SWPM tool and usually display great efficiency in managing bug reports. Decisions about how the social order should evolve in the context of a particular bug report are usually made quickly, one exception being instances of negotiation regarding where responsibility for fixing a software problem lies. Reasons for the community’s efficiency are that the repository is simple to use and the SWPM process used by this community is consistent with SWPM processes used in other software development settings.

Issues and questions related to social order occur at several levels of analysis in this community. This chapter looked closely at social order at the bug report and bug report network levels. Social order has not been examined at either the product / component level or at the overall F/OSS community level in this project.

A number of interesting and important questions remain for further study. A number of questions relate to the patterns of social order that may occur throughout the bug report repository. Are there common DCE’s that re-occur in the repository, especially in terms of the individuals who take the formal roles of reporter / assignee / QAContact? Are there individuals who consistently play a particular formal or informal role in a large percentage of bug reports? Do individuals take responsibility for bug reports across a variety of products / components or do they tend to “specialize” in one
portion of the system? Can the evidence from the bug report repository reveal information about central figures in the F/OSS development community?

Other questions remain about the motivations of the participants for the actions they take in regard to the establishment and evolution of social order. The content analytic approach of this project makes it impossible to address such questions at this time.

There are a number of questions remaining about the relationships between social order and information practices that could not be addressed within the scope of the current project. Do the information practices engaged in by community members vary as a function of their formal or informal role? Is there a more complex relationship between role, information practices, and the software problem management phase (e.g., identification and reporting; triage; expert analysis; verification and closure)? For example, if information practices include activities like searching, describing, evaluating, comparing, and planning, do assignees tend to engage in different information practices depending upon whether the bug report is in the triage or expert analysis phases? It is also possible that there are other activities that have an effect on the establishment and evolution of DCEs and related information practices. Also, there are other questions about how the outcomes or findings from particular information practices, like searching, can affect the evolution of social order at the bug report level.

Finally, the analysis here provides a basis for quantitative examination of the possible correlations between factors related to social order or DCEs and community outcomes. One example of a relationship that could be examined in this way is the correlation of the size or complexity of the DCE and time to resolution for particular bug reports.
CHAPTER NINE: DISCUSSION

Introduction

This project examined the information practices in use within a large, successful distributed work community. The community investigated here is a Free / Open Source software development organization that relies upon use of both custom and ubiquitous information and communication technologies to support its efforts to manage software problems. I analyzed a sample of bug reports drawn from the bug report repository maintained by this large, successful, distributed F/OSS software development community. The research questions motivating this project were (1) How do F/OSS software development communities manage software problems? (2) What varieties of information and activity occur as part of software problem management? and (3) How are information, activity, and social order related in software problem management?

This case study employed a holistic, social informatics approach, using software problem management (SWPM) work as an exemplar of distributed work to address questions about the interactions between information, activity, and social order, and to help us understand how distributed organizations utilize information to actually “get things done.” This project applied ideas from research into human information behavior, negotiation, collaboration, sensemaking, and software engineering in order to understand the information practices of this distributed software development organization. While research into software defects has been done, primarily from a software engineering or psychological perspective, this project was unique because it looked at software problem management as a complex, organizationally situated socio-technical process. This project took a particularly close look at the information-related behavior of individuals and small groups in the context of individual bug reports and bug report networks (BRNs) in order to identify and understand the information practices used in distributed work.

This project makes methodological, theoretical, and direct practical contributions to our understanding of information practices, distributed work, and software
engineering. This project contributes to methodological technique by exploring application of content analysis and grounded theory to the examination of distributed work in SWPM, a sub-domain of software engineering. It provides theoretical advances by building explicit links between investigations into human information behavior and information practices, traditional research concerns of Library and Information Science, and detailed, ethnographic studies of computing in the workplace, a traditional concern in computer-supported cooperative work, participatory design, distributed collective practices, social informatics, and related traditions. This study found numerous cases of the mutual influences of activity, information, context, processes, social order, which are all factors in individual and collective information practices. This study also enriched the SWPM models published by other researchers (e.g., Carstensen, Sørensen & Tuikka (1995); Crowston (1997)). Finally, the findings reported here are being applied to the development of computational tools to support scalable analysis of similar corpora and to improve SWPM practice.

A systematic random sample of 385 bug reports was drawn from a collection of more than 182,000 bug reports held in this community’s bug report repository. A number of additional bug reports were analyzed after they were selected from the repository using the theoretical sampling technique from grounded theory. The bug reports in the sample were treated as texts, using content analytic approaches and grounded theory procedures to support the development of categories of phenomena warranted by the data and new and revised theoretical constructs.

The data analysis revealed a number of large- and fine-grained components in the SWPM domain that could be the subjects of research, ranging from society at large, F/OSS meta-communities, and individual F/OSS communities. The meso-grained units of analysis focus on information and activity within this particular F/OSS community and rage from the community’s software development practices, their SWPM practices, the bug report repository in use, and the range of first-class and virtual information objects contained in the repository (e.g., bug report networks, bug reports, bug report state sequences). Fine-grained units of analysis are those that are typically subcomponents
of bug reports and BRNs, such as comments, dialogs, statements, and instances of negotiation. This project was concerned primarily with meso- and fine-grained units of analysis, with the bug report being the fundamental unit of analysis.

The distinction between bug reports, a first-class information object, and software problems, which are phenomena encountered during situations of software-in-use, was identified and discussed. This distinction is important and must be kept in mind throughout the analysis in order to ensure that the analytic language is precise and clear. Bug report networks (BRNs), identified and described in detail here for the first time, provide an excellent view into the information practices of this distributed community. BRNs are dynamically composed as community members assert a variety of formal and informal relationships between individual bug reports as a result of the community’s various information practices, including searching, comparing, and analyzing information, as well as its enactment of its SWPM process.

Layers of context, including structural contexts (technical, economic, organization, and political / ideological contexts), meso-level contexts (the bug report repository), and micro-level contexts (bug report networks, bug reports, and negotiation contexts) were also identified and explored in the course of this project. Accounting for context remains a challenging issue for researchers in information studies and social informatics generally. This project contributes a means of categorizing layers of context in order to support future empirical research and development of theory about information practices by individuals, groups, and organizations.

The normative SWPM model, based upon previously published research (Carstensen, Sørensen & Tuikka, 1995; Schmidt & Simone, 1996; Crowston, 1997; Kajko-Mattsson, 1999) was presented in Chapter One and enriched based upon the evidence drawn from the bug report sample analyzed in this project. Four distinct phases of the SWPM process were identified and contrasted with the more deterministic “engineering process oriented” descriptions presented by previous researchers. The four phases of SWPM were labeled Problem Identification and Reporting; Bug Report Triage; Expert Analysis, Fixing, Testing and Deployment; and Bug Report Verification and
Closure. Each phase is identified as either predominantly an engineering or sensemaking phase. Two additional sensemaking sub-processes, collective debugging and individual debugging, were also identified. Negotiation, a basic social process, was identified as a key part of all phases and sub-processes (although instances of negotiation rare during problem identification / reporting).

Social order is another topic that is best understood in terms of the unit of analysis currently in focus. The F/OSS community as a whole has a relatively stable, fixed social order, but individual bug reports and bug report networks (the primary focus of this project) are characterized by fluid, dynamic social order. The concept *distributed collective ensemble* (DCE) helps us see the dynamic, situational ways in which the distributed participants (reporter, assignee, QAContact, and those community members placed by themselves or others into less formal roles) organize in response to activity, negotiation, and evaluation of evidence in support of SWPM.

The remainder of this chapter summarizes the main findings from this empirical investigation into the distributed information practices of this F/OSS development community as they manage software problems. The findings from this project build upon the following theories and research approaches described in detail in Chapter Two:

- Human information behavior and information practices
- Theories of negotiation, context, and negotiated order
- Sensemaking (Weick, 1995)
- Coordination / cooperation theories

These established theories and approaches are used to help interpret the data, create the map of SWPM work, and interpret that map. Concepts from human information behavior research are used in identifying the information objects present, used, not used, and created as the community manages software problems. Several theoretical constructs associated with the work of Anselm Strauss and his colleagues are employed throughout the analysis to help describe and explain the phenomena found in
this community’s bug reports and the interrelationships between information, activity, context, process, and social order. Principal theories from Strauss applied in this project include negotiation, negotiated order, contexts, and to a lesser extent, trajectories. Sensemaking, in Weick’s sense of the term, is used as an over-arching conceptualization of the process of managing software problems. Theories of coordination and cooperation are used in particular to help explain the mutually constitutive roles of technology and process as well as to help understand the how the community dynamically organizes actors in this distributed environment. Following this, the implications and impacts of the findings are discussed. Finally, directions for future work in software problem management (SWPM) are considered.

Summary of Contributions

This section summarizes the contributions made by this project to the research questions, stated in Chapter One, about how software problems are managed by distributed F/OSS development communities, the kinds of information practices employed in SWPM, and the relationships between information, activity, and social order in distributed work.

Mapping the Domain of SWPM

As noted in Chapter One, one of the primary objectives of this project was the development of the first in-depth, detailed empirically grounded description of the information practices employed in this distributed SWPM domain. This objective has been met through the analysis of the sample of 385 bug reports presented in this report. As noted earlier (in Chapters One and Two) the most detailed existing descriptions of SWPM were normative high-level descriptions of the process as an engineering process. Carstensen, Sørensen & Tuikka (1995) provided the richest description of those available, including important details on the kinds of information, information formats, and social
and organizational context. Schmidt & Simone (1996) use Carstensen et al’s research as a key example in their seminal explication of coordination mechanisms. Crowston (1997) (and, to a lesser degree, Kajko-Mattsson, 1999) provides a straightforward business / engineering process description that contains relatively little detail about the information, processes, contexts, and social order of the organization he describes, and in the end focuses the paper on the definition of coordination theory.

Chapter Four describes the main components, primarily units of social organization and information objects, which are reflected in the bug report data. Related information about this community and its SWPM practices, and the larger worlds of F/OSS development and software engineering are also discussed. The hierarchy of analytic units presented in Figure 4-1 summarizes the units of analysis that appear to be the most important in the context of this community’s SWPM work. The hierarchy of analytic units forms one fundamental dimension of the map of distributed work in this domain. This hierarchy places the elements that the remainder of this report focuses on, such as bug reports, bug report networks, and instances of negotiation, in a more complete context of social units and information objects. It also illustrates the rich, complex nature of the information artifacts and other potential units of analysis in this domain and provides future researchers with a guide they can use to investigate similar domains at varying levels of granularity.

The fifth chapter clarifies the important distinction between the phenomenon of a software problem and its representation as an information object, a bug report. This distinction is critical because it allows the analysis presented here to clearly demonstrate the many fundamental ways in which SWPM is simultaneously information management and the resolution of software problems. The chapter also provides the first description of bug report networks provided in the research literature. BRNs are frequently created as community members assert formal and informal relationships between bug reports. Linking bug reports together is identified as one of the fundamental information practices employed by this community to support their SWPM process. About two-thirds (65%) of the bug reports contain one or more formal or informal references to other bug reports.

230
BRNs represent (1) information ordering strategies that support collocation of related bug reports, decreasing cognitive and organizational effort of the community members; (2) sense-making strategies wherein BRNs provide more refined representations of software and work-organization; and (3) social ordering strategies that rearrange collective relationships among community members.

Bug report networks (BRNs) are an exemplary context in which the observer can see direct relationships between information, activity, contexts, processes, and social order. Consider the activity of linking two bug reports together to form a BRN. This activity can take place during the problem reporting, triage, or expert phases of the SWPM process. Instances of negotiation, a basic social process, can occur during various phases of the SWPM process, for example, to ascertain whether or not a new bug report is a duplicate of an existing bug report. By convention, the problem reporter should search (another activity) for duplicate bug reports describing the same phenomenon before creating a new bug report. The act of linking bug reports together creates a new information object, the BRN. The ability to create formal links between bug reports and create BRNs is a function of the technical context provided by the community’s bug report repository. A new context, the BRN context, is created when the linkage is established between the bug reports. Finally, a new distributed collective ensemble (DCE) is created when a bug report is created, and multiple DCEs are combined when bug reports are linked together, demonstrating how the activities of the community members during information management and correction of software problems affect social order.

Although this project used bug reports as the primary unit of analysis, the map of the domain provided here provides insights into other units of analysis that have the potential for yielding important insights into SWPM in particular or distributed work in general. More detailed discourse analysis could be performed to more clearly identify patterns of communication that occur in the bug reports as the participants manage the information objects and the problems they represent. This report identified negotiation as one common and important such processes, but other patterns, such as question and answer sequences or hypothesis generation and testing, are amenable to more detailed
analysis. Bug report state sequences, described in Chapter Four, may contain information that can help predict the outcomes of the SWPM process, such as successful resolution of a problem or elapsed time to resolution.

**SWPM Is Both Information Management and Correction of Problems**

One of the most important insights from this project is that SWPM is fundamentally information management. SWPM is primarily information management in the sense that information management activities are generally antecedent to the actual correction of the code and thus to the resolution of the problem, at least when the creation of bug reports occurs (cases where a programmer detects and corrects a bug during coding without creating a bug report are common but not represented in the bug report repository). While the overall goal of SWPM is correction of problems and the improvement in the quality of software, previous examinations of this work have overlooked the importance of information management in meeting these overall goals (Carstensen, Sørensen & Tuikka (1995) is an exception to this shortcoming).

The entire SWPM process used by this community is driven by the data contained in the bug reports and the affordances of the bug report repository. The bug report repository provides a place for the people encountering software problems to communicate to the people who have the skills and means to correct software problems. The reporter makes a determination that the cost in effort of reporting the problem is exceeded by the benefits of possibly seeing the problem resolved. Bug reports are the fundamental unit of work in this domain and are first class information objects. As information objects, bug reports support formal and social processes established by the community to facilitate SWPM. In this community, the SWPM process (see Table 7-1) defines terms and conditions for managing both the information objects and the software problems, although the distinctions between working on the information objects (information management) and the underlying problematic phenomena (changing the code to correct problems) are not always drawn clearly by the community members. For
example, you often see language like this: “Could be dup of bug X or bug Y” (BR 35) when the person making this comment is actually referring to a duplicate relationship between the current bug report and bug reports X and Y.

Several factors influence the need for concerted information management within the SWPM process. One factor is the size of the system and the development community. The bug report repository maintained by this F/OSS community contains, at this writing, more than 250,000 individual bug reports. It would not be surprising to find a correlation between the scale of the system being developed or the size of the development community with increasingly sophisticated SWPM information management. Another factor is the level of sophistication of the software development organization itself. Consider, for example, the Capability Maturity Model (CMM) described in Chapter Two, which identifies the adoption, use, and documentation of processes to assure quality, control system change, and prevent defects as indicators of the maturity of software development organizations (Herbsleb et. al., 1997). A third factor is the potential cost to the organization of failing to manage software problems in a way that supports visibility of the problems, the activities of the community in response to the problems, and the accountability of individuals for bringing problems to resolution in a timely manner.13

The community’s information management work is done in support of fixing software problems. Because SWPM is a community process, SWPM requires articulation work: “all the tasks need to coordinate a particular task, including scheduling subtasks, recovering from errors, and assembling resources” (Gerson & Star, 1986) (p.258). A bug report is a standardized information format arrived at through an evolving, negotiated process. The bug report supports the work of the community as it enacts its collective SWPM process. It is designed to simultaneously support the idealized SWPM process typically presented in the literature (see, for example, Crowston, 1997) and the

---

13 My own professional experiences indicate that small, collocated development teams (fewer than ten individuals) in low-stakes settings have far less motivation and less to gain by using a formal process and complex information management strategies than large, distributed, cross-organizational development teams working in safety-critical or other high-stakes environments.
unpredictable situations that arise as real bug reports are created and handled by the community.

The bug report repository and the bug reports it contains form a *coordination mechanism*, “a construct consisting of a *coordinative protocol* (an integrated set of procedures and conventions stipulating the articulation of interdependent distributed activities) on the one hand and on the other hand *an artifact* (a permanent symbolic construct in which the protocol is objectified) [emphasis in original] (Schmidt & Simone, 1996) (p.165-166). The bug report is the artifact in this situation, although other first-class and virtual artifacts are present in this community’s SWPM process (e.g., bug report networks; comments contained in the bug reports). The bug report is an open, public information object that also makes the responsibilities of the community members visible and clear to the entire community. Openness and visibility of the bug report is essential to support this distributed collective SWPM process which, in this and most other software development organizations, relies upon division of responsibility / duties in order to stress individual and collective accountability and thus help ensure quality of repair.

The coordination mechanism in use here also serves to make routine, through pre-computation, much of the articulation work that must be performed when a bug report is created (Schmidt & Simone, 1996). For example, default assignees and QAContacts are determined by the classifications assigned by the reporter or person performing triage (e.g., the fixed fields “product” and “component”). Notifications are automatically sent to the assignee and QAContact when they are associated with a bug report, reducing certain costs of articulation, like communication and notification. Changes to these values classification values are recorded as part of the bug report change log and included in the repository. Change log entries contain the date and time of the change as well as the identity of the individual who made the change.
**SWPM is a Distributed Collective Practice**

The term *distributed collective practice* (DCP) is used to denote “activity distributed across geographical and conceptual distances, time, collective resources, and heterogeneous perspectives or experiences. DCP relies on information infrastructures and technologies…” (Distributed Collective Practices, n.d.). Distributed collective practices are distinguished by their lack of reliance upon a central authority or central actor to decide or determine what work should be done, which decisions should be taken, or the sequence in which activities should occur. In this domain, the sum of the contributions of the many parties forming the distributed collective ensemble (DCE) associated with each bug report (the reporter, the QAContact, the assignee, the commentators, etc., as described in Chapter Eight) continuously accrete and are evaluated on a situational basis. While the defined SWPM process and the affordances of the bug report repository reify a particular general trajectory for managing software problems, the trajectories taken by each bug report vary widely (see, for example, Table 4-4, the 101 distinct bug report state sequences occurring in the sample of 385 bug reports) and cannot be predicted at the time the bug report is created. The process of continuous contributions, accretion of information and knowledge, and continuous evaluation of evidence leads to activity by the community regarding management of the bug report (the information object, or artifact) and the resolution of the software problem.

SWPM in this domain is clearly distributed: the community of users and developers of the products of this F/OSS development community are scattered through the world, and, unlike traditional software develop organizations, the developers are neither collocated nor bound by a formal relationship to an employer. The SWPM process relies upon moment-to-moment contributions of new information and event-by-event evaluations and decisions made by the members of the bug report’s DCE. While certain actions and decisions are reserved for particular individuals playing particular roles (e.g., QAContact or code reviewer), there are many individuals in the community as a whole who are capable of fulfilling each of these roles.
The SWPM process is largely asynchronous and distributed with respect to time, without the tight coupling of activity that is typical in other domains, like real-time supervisory control of infrastructure. SWPM here is clearly collective: with very few exceptions, the roles of reporter, QAContact, and assignee are played by separate individuals. In many cases (see the examples discussed in Chapter Eight) the DCE is comprised of more than three individuals.

The affordances of the coordination mechanism, including the bug reports and the SWPM procedures, comprise the critical infrastructure upon which this community relies for the management of software problems.

**SWPM is a Distributed Sensemaking Enterprise**

The entire SWPM process in this community can be interpreted as a distributed sensemaking endeavor. The bug report itself is a frame of reference for the community: it provides a common format for reporting and managing bug reports and associated software problems that is simple and generally well understood. It is simultaneously simple enough for a newcomer to understand (its structure is similar to all SWPM repositories and reports, even paper based systems like that described in Carstensen, Sørensen & Tuikka (1995)) yet sophisticated enough to support those who participate in the triage, expert, and verification phases as assignees and QAContacts.

Weick (1995) describes generic sensemaking as a strategy for an organization or community to reassure themselves “the world makes sense and things are under control” (p.170) through interlocking routines. Generic sensemaking supports an organization’s need for “swift socialization, control over dispersed resources, legitimacy in the eyes of stakeholders, measurable outcomes, and accountability” (p.170). However, sensemaking organizations also evolve their routines when occasions arise where routines are not sufficient. Thus, a sensemaking organization’s “habituated’ action patterns and ‘routines’ are not completely automatic” (p.171).
The sensemaking process in SWPM begins with the encounter by a human of problematic or surprising system behavior. In some but not all cases, the person creates a new bug report and provides some evidence about the problematic phenomenon. In many cases, aspects of the process are contingent and, as we have seen, subject to negotiation. Information gathered and associated with a bug report can be analyzed, but decisions must be made about what actions should be taken next, what information is needed next, and who should take those actions. Each bug report created by this community is an information object that by its nature, its context and the coordination mechanism within which it exists requires a collective community response. The coordination mechanism and coordinative protocol (Schmidt & Simone, 1996) (and see also the SWPM process detailed in Table 7-1) embodies what the community needs for SWPM to appear to be a routine and predictable process.

The structure of the bug reports and the affordances of the bug report repository support accountability by naming participants and defining roles and associated privileges. They also provide a firm and widely understood foundation for the socialization of new community members because the repository and its contents are modeled upon the SWPM processes commonly used in all software development organizations. The naming and identification of participants also helps to manage the scarce human resources needed to manage bug reports and solve software problems, largely through the repository’s information management capabilities and its support for articulation work.

The bug reports are information objects that represent the problematic system behavior reported by a community member. The data added to the bug report after its creation represent some of the community’s activities and some of the information found, created, or offered by community members in their effort to make sense of the problematic phenomenon and generate consensus on issues of system behavior, problem causes, values (e.g., determining if the bug report represents a software problem or not; determining the appropriate priority level in the community’s queue of “things to fix”), the appropriate software design, and how the code should be fixed.
Negotiation, a basic social process discussed in Chapter Seven, is one of the ways in which community members seek and build consensus on (and make sense of) these issues before the bug report and its underlying problem can be considered “resolved.” In terms of consensus on behavior, problems are much easier to fix if the community understands unambiguously how to make the problem occur. Negotiation is likely to occur until the behavioral issue has consensus, and little will be done before the problem is deemed “reproducible.”

Value issues may be subject to interpretation and negotiation, particularly issues like “how important is this problem,” “is there really a problem or not,” “what is the appropriate priority for this bug report,” or “has this problem been adequately fixed.”

Causation may be negotiated any time from the creation of the bug report (the reporter, if he is knowledgeable or if she is also a system developer, may propose possible causes) until the expert analysis, fixing, testing, and deployment phase is completed.

Design issues, particularly regarding what is the best approach for fixing the problem represented by the current bug report, are one of the most frequently negotiated issues (see Table 7-5). In some cases, debate on the appropriate design can last for months, involve numerous participants taking several turns. Extended negotiation on these topics is likely when the issue is a modification to the user interface or how to respond to requests for new features, or enhancements, to the system.

Finally, the code itself is frequently an object of this consensus building process. The key example in this community is the expected review of code patches submitted by developers as corrections for software problems. The defined process for “checking in” fixes includes two separate code review by a “code reviewer” and a “super reviewer.” Code reviewing is a common and basic software quality assurance technique employed by many organizations and has been demonstrated to be one of the most effective way of ensuring that software (and a bug fix) is well designed, conforms to standards, is implemented efficiently, is understandable, etc. Bug reports often (but not always) contain evidence of code reviews. A code reviewer, when he has comments on the submitted patch, frequently enters those comments into the bug report for the assignee
to see and respond to. The patch cannot be committed to the code tree until the code reviewer and the developer reach consensus on the quality of the code.

**Distributed Collective Information Practices Are Fundamental**

This project examines the collective information practices of this F/OSS development community, which are fundamental to the effectiveness of its SWPM process. This project also improves our understanding of human information practices at an individual level and at a collective level, as well as distributed collective information practices at the most general level. What has been learned as part of this project contributes to models of information practices in both co-located and distributed settings.

SWPM is complex, information intensive work. Chapter Four points out the many social, organizational, technical, and information components at play in this field of work. The complexity of the software produced by this community, the rate at which bug reports are created by the community (hundreds per day), and the size of the community necessitate collective management of the software problems. Chapter Seven, on process in distributed work, illustrates many of the ways in which the information practices of this community are collective. The activities of individuals, whether they are testing software, evaluating system behavior to decide whether a problem exists or not, searching for a bug report that may already sufficiently describe the problematic behavior they have encountered, creating a new information object (a bug report) if necessary, evaluating the new bug report to determine if it is reproducible, invalid, or a duplicate, or has some other relationship to existing bug reports, and so on, combine together to generally move bug reports successfully through the SWPM process.

The LIS tradition of human information behavior research has begun to incorporate wider conceptions of users, human information behaviors, situations of use, contexts, and types of information than was typical of the period preceding Dervin & Nilan’s seminal paper (1986). Examination of information use and the creation of
information are just beginning to be incorporated in studies of this type. Bishop et. al. (2003) and McKenzie (2003, 2004), in particular, acknowledge the centrality of collective or community behavior in their adoption of a holistic view of human information behavior.

This project extends Wilson’s (1999) nested model of individual human information behavior and builds upon McKenzie’s (2003) examination of the information practices of people in co-located group or community settings by moving from a focus on the information behavior of individuals to the collective information practices of groups, organizations, and communities. Another layer has been added to this model by considering the more complex issues of distributed information practices, which are the object of the current project (see Figure 2-1).

Information seeking and use of computer-based bibliographic retrieval systems has been a standard area of study for decades in Library and Information Science and is typical of the activities emphasized in research referred to by terms such as human information behavior research, human-information interaction, or information needs and uses. Much of this work has been limited in scope, concentrating on individuals using a single system in library, laboratory, or educational settings. Far less attention has been given to searching for other kinds of information in a wider variety of settings (e.g., how individuals search within organization-specific databases, such as a bug report repository, in support of processes like SWPM). The information practices perspective broadens the examination of the use of information, including information seeking, by considering all the activities, information, and processes employed by individuals, groups, and organizations to locate information. In this project, for example, it is apparent that a common and critical activity, searching for and identifying duplicate bug reports, is a distributed, collective information practice. It is not solely the responsibility of any one individual, nor the responsibility only of a group of individuals fulfilling a certain role (e.g., QAContact). The person reporting the problem, the person “triaging” the bug report, and the software developers trying to solve the problem all participate in identifying duplicate bug reports. The information practices approach can be applied
beyond information seeking and searching to phenomena like information compounding, linking information objects together (e.g., to form BRNs), and so on, in distributed settings like the one studied here.

Coordination mechanisms are always under-specified (Schmidt & Simone, 1996) because the “the nominal preconception cannot encompass and denote the infinite multiplicity of actual circumstances and occurrences unfolding during its situated enactment” (emphasis in original) (p.175). The bug report repository used here is underspecified in this way, leaving enough “slack” in the system for its users to make adaptations on the fly in order to better fit the repository to the emerging and unanticipated constraints and conditions of the work. The adaptation whereby community members define “meta” bug reports is one example of how under specification is an advantage for the embedding community.

Distributed Work is Shaped and Influenced by Interrelated Layers of Context

The situated investigation of information practices can be a daunting task and there are many challenges in performing research of this kind. For example, how do you effectively take account of context and the uniqueness of each setting? If every situation is unique, to what extent is it possible to generalize from specific results? As a case study, this project cannot conclusively demonstrate the ways in which context influences the performance of distributed work: only comparative studies that control for specific contextual properties can do that.

However, one limited comparison can be made between the layers of context described in this report (see Chapter Six) and the setting described by Carstensen, Sørensen & Tuikka (1995). Carstensen et. al. provide a detailed description of the setting, the software development project, and the paper bug report form used to manage software problems. The SWPM process used by the community studied here would be far different in character and, in all probability, effectiveness, if, instead of relying on a Web-accessible relational database management system with asynchronous computer
mediated communication (e-mail notifications), the repository were paper based. This development community is truly globally distributed, so imagine the delays and costs in both reporting and notification if reporters needed to submit paper forms and, upon problem assignment or re-assignment, a form had to be mailed to the assignee on another continent. It’s clear that this community could not operate in nearly as efficient a way under those conditions if it had to rely on a paper based SWPM process.

The contextual differences between these two settings are (1) collocation of developers and engineers as opposed to global distribution of developers and other community members; (2) scale of the community, 50-100 participants as opposed to tens of thousands of active community members; (3) the coordination mechanism, specifically the bug report repository used to support SWPM; and (4) the differences in structural contexts (e.g., a for-profit firm versus a largely volunteer-driven F/OSS development project). It is likely that, at the micro-level, many aspects would be similar, such as the bug report formats, the division of responsibilities among reporter, assignee and QAContact, and the occurrences instances of negotiation within bug reports, to name just a few.

**Future Directions**

This project has addressed how information, activity, context, process, and social order are related in distributed work, using the domain of software problem management as an exemplar of distributed work. The findings from this project provide data and signposts for continued research using the existing setting and data, using other settings in the same domain (SWPM), or in other settings in other domains (e.g., other settings where (1) problem management is a key concern, whether the setting is primarily distributed or centralized; (2) management, creation, and use of diagnostic information is a central concern; (3) any type of distributed, information-related activities occur). Examples of other settings and domains include medical diagnosis, traditional
bureaucracies (e.g., government, education), digital libraries, and real-time supervisory control of distributed infrastructure.

There is still potential to learn from this setting and its SWPM process. Some issues that could be explored in greater depth within this same F/OSS development community, using data from the same bug report repository include: bug report networks (Chapter Five); the role of negotiation in SWPM (Chapter Six); and the significance of bug report state sequences (Chapter Four).

**Future Research Program**

**Bug Report Networks**

The nature, qualities, and effects of the community’s creation of bug report networks (BRNs) are some of the key findings of the current research project. The analysis performed so far demonstrates that bug report networks are common: 65% of the bug reports sampled are part of a BRN. Members of this community commonly use both the formal, symmetrical relationships of duplication and dependency as well as a wide variety of informal relationships as discussed in detail in Chapter Five. The work done so far has raised a number of other questions to be pursued in the future:

- Identify the situations in which BRNs are helpful (or unhelpful) in managing software problems; can we understand the extent to which large, complex BRNs are taken into account by community members with different roles in the community? Are BRNs of greater or lesser importance to the community during particular phases of the SWPM process (e.g., triage vs. expert phase)?
- What are the motivations for asserting various formal and informal relationships between bug reports? What advantages accrue to the
community and to individual community members when these relationships are asserted?

- Are BRNs a phenomenon present in all bug report repositories? How do the capabilities of different repositories or the conventions developed by a SWPM community influence the use of BRNs (one common convention in this community is the use of the keyword “meta” to indicate that the bug report serves as the central bug report in a network of dependent bug reports)? Are other formal types of inter-bug report relationships supported in other bug report repositories?

- What is the range of complexity of BRNs in this and other bug report repositories? What are the most useful metrics for measuring the size, character, and complexity of BRNs? For example, a BRN can be thought of as a graph, with each bug report as a vertex in the graph: is this a useful analytic technique? Is it possible to reliably identify the “central” bug report within a BRN (this is not always obvious because interconnections between bug report networks seems to be quite common)?

- What are the most useful representational forms for bug report networks? Can visual representations (e.g., Figure 5-1) contribute to our understanding of BRNs? Can better BRN representations contribute to improvements in a community’s SWPM practice?

- How does the inclusion of a bug report in a BRN affect the community’s ability to move the bug report to a terminal status (e.g., resolved; closed)? Is there a correlation between BRN membership and problem solving performance (e.g., time to resolution)? Is the correlation positive or negative; or does the correlation vary depending upon other factors?

- What are the relationships between the creation and evolution of BRNs and time? Are there recurring patterns of evolution with respect to time, or with respect to the different phases of the SWPM process?
Automatic extraction and representation of bug report networks may be an important part of addressing the questions raised here. Two questions regarding the automated extraction and representation of BRNs also remain to be addressed:

- What are the best techniques for automatically extracting BRNs from a bug report repository in a reliable manner, particularly in terms of identifying informal relationships?
- Is it possible to develop computational tools to discover latent, or unasserted, relationships between bug reports? For example, in this repository, more than 40% of the bug reports are involved in a duplicate relationship, and currently all identification of duplicate relationships is done manually. An effective tool for identifying and suggesting potential, yet undiscovered, relationships – both duplicate and dependencies – might have a positive impact on the productivity of the community.

Basic Social Process: Negotiation

Negotiation is a basic social process that occurs frequently in virtually all social settings, including work settings like the one studied here. The bug reports reveal that negotiation is pervasive in SWPM, and that negotiation occurs throughout the entire SWPM process. A given instance of negotiation frequently involves more than one participant and more than one “turn.” Thus, one factor that might affect time to resolution is the presence of instances of negotiation.

Questions about the relationships between negotiation and the roles of community members remain to be investigated.

- Who (that is, which role(s)) typically negotiates (reporter; assignee; commentator; QAContact)?
• Are there patterns regarding peoples’ roles and typically negotiated issues (e.g., QAContacts often negotiate about whether the problem is fixed or not, but rarely negotiate design alternatives)?

• Are there significant variations in the issues negotiated during the different phases of the SWPM process?

Studies that compare the findings from this domain with the findings in other domains could help determine whether patterns of negotiation do or do not correlate with patterns in information practices. Which information practices, for example, are more or less likely to include instances of negotiation? Is negotiation on particular topics more likely during certain information practices?

Information Object State Sequences

One of the goals of the wider efforts to understand SWPM and software quality assurance is to develop useful, predictive models that can be applied to the improvement of software practice. A key contribution of this particular project is the identification and exploration of factors that can, for example, serve as the independent variables in such models. But the sort of time-consuming, human coding and analysis performed in this project cannot scale to support analysis of more than a small fraction of the records in this corpus or to support similar efforts in other communities using the same bug report repository system.

Chapter Four provides a description of the state sequences and values in two fields found in every bug report contained in this community’s bug report repository, the bug report status and resolution fields. In this domain, and within the bug report repository used here, there are a finite set of states that bug reports can traverse from the time they are created until they are no longer active. The qualitative analysis here shows that there are small number of regularly occurring state sequences and a larger number of rare or unique state sequences: 101 distinct state sequences occurred within the sample of 385 bug reports. This project made provisional distinctions between “normative,” “sub-
optimal,” and “deviant” state sequences. What remains to be done is further analysis of a larger sample (or on the entire corpus) to determine whether the qualitative distinctions made here have any practical import on the SWPM practices of the community.

Some modeling of these state sequences has been performed using finite state machines and Markov models by other participants in this project. This work holds promise as a method for the automatic extraction of process models (Gasser & Ripoche, 2003). Other research questions remain regarding the bug report state sequence data:

- Is it possible to correlate “deviant” and “sub-optimal” state sequences, or sub-sequences, with other data available in the bug report (such as membership of the bug report in a BRN; evidence of negotiation over particular issues; number of individuals in the bug report’s DCE; severity / priority; etc.) and build a reliable, automatic “early warning” system to identify salient bug reports to community members?
- What are the relationships between state- or sub-state sequences and time to resolution?

Deeper Investigation of Information Practices

This project is concerned with information practices of individuals and groups in the contexts of this software development organization. A number of information practices were identified, but it is not clear if they are all of equal significance, or if their level of significance varies depending upon other factors. Each of the identified information practices, and there may well be more that have not yet been identified, warrants further investigation to both better understand distributed work in this setting, but to also contribute to the larger goal of developing a more complete theory of information practice. Relationships between information practices and other factors, like the current phase of the SWPM process or the role of the actor, may yield insights useful for both the development of models and theory as well as the design of practical tools.
One particular relationship that can be studied is the relationship of specific information practices to the phase of the SWPM process within which it is employed. One example is the comparison of evidence in the bug report to the behavior of the software. During the identification and reporting phase, what are the goals, inputs, activities, and outputs/outcomes of comparing the system’s behavior to data contained in the bug report? During triage, are there significant differences in goals, inputs, activities, and outcomes in the comparison of evidence when compared to identification and reporting? How does this information practice during the expert fix, test, and deploy phase compare to the same information practice performed in the earlier phases?

Another relationship to explore is that between information practices and the role of the current actor. For example, if the reporter, QAContact, and assignee are all heedful of the assignee’s activities during the fix, test, and deploy phase, do they (1) engage in different information practices altogether, depending upon their role, or do they (2) engage in the same information practices, and, if so, are their performances of these practices similar or different?

Additional analysis is warranted regarding individual information practices in order to more fully define and understand them. In the LIS literature, a few human information behaviors have been studied in depth (e.g., searching for information and browsing for information), but some information behaviors, like comparing or evaluating evidence, have not. Each distinct information practice can be analyzed in terms of its required and contingent inputs; its processes or activities; its contexts; and the outcomes.

A higher level analysis of the relationship of individual information practices on the community’s outcomes should also be considered. For example, does engaging in searches for duplicate bug reports correlate to increases or decreases in time to successful resolution? If more effective tools to support identification of duplicate bug reports, whether are automated or not, are provided to the community, is the community’s performance in fixing software problems significantly improved?

Thorough investigation of individual information practices may also provide direct practical benefits to system designers and users. Improved understanding of the
details of an information practice might lead to improvements in system design, system usability and the community’s SWPM performance.

Distributed Collective Ensembles

The discovery of evidence that the routine activity of community members has a direct and vital impact on the community’s social order is another key finding from this project. Actions such as (re-)associating a bug report with a particular product value or asserting a formal relationship between bug reports results in social reconfiguration. These effects are important at both the level of individual bug reports and in the context of BRNs. The section above listed some outstanding research questions related to information practices and the formal roles community members play in terms of particular bug reports, but similar questions remain regarding the information practices of community members in the informal roles, such as commentator, articulator, facilitator, and observer, identified in Chapter Eight. The remaining informal role, reader, is currently impossible to analyze given the design of the bug report repository and the kinds of usage data it collects, but it is possible to modify the system’s design in order that it capture and record the number of times a bug report is returned in results set and the number of times it is accessed by a reader. If such data existed, we could address questions about the significance and contributions of readers to the overall performance of the community in solving software problems.

Other questions about the composition of DCEs in this community remain to be addressed. For example, are there particular pairs or sets of individuals who frequently co-occur in DCEs? It would also be interesting to trace the behavior and contributions of individuals over time and across all of the bug reports to which they contribute or with which they are associated.
Context and its Relationship to Distributed Work

The issue of context was addressed by this project, with a layered context model presented as one way to make this difficult aspect of situated research into information practices tractable. The layers identified were structural, meso-, and micro-level contexts. Issues of context are vital in research and in terms of providing practical support for the information practices of any community. Context is what constitutes the situational, the unique aspects of information practices. This project does not provide prescriptions for researchers or practitioners concerned with information systems or practices. However, the analysis presented here may be employed in future research on context:

What similarities and differences exist between the structural, meso-, and micro-level contexts in other F/OSS development communities? Do characteristics of the community, such as size, complexity, maturity, predict which aspects are similar and which are different? What are the similarities and differences between F/OSS communities and traditional development communities? Between software development and other communities (e.g., infrastructure design and control; medicine; digital libraries)?

Process and Distributed Work

This project identified qualitatively different kinds of processes at play within this community as it performs SWPM. Some processes and sub-processes can be characterized as rational engineering or business processes (e.g., like the high level, enhanced normative SWPM process discussed in Chapter Seven can be evaluated as such a process). Other processes and sub-processes can be more accurately characterized as sensemaking processes (e.g., triage; analysis, fixing, testing and deployment; individual and collaborative debugging). Other processes and sub-processes identified are more accurately characterized as basic social processes, such as negotiation. Other processes, like community re-enactment and voting, were identified but not analyzed in detail.

Research questions regarding process that remain include:
To what extent is the layered process model described in this report applicable to other SWPM communities? Are there significant differences between F/OSS and traditional software development communities with regard to process?

Which, if any, of the elements of this process model are transferable to other domains, such as software engineering in general; medical diagnosis; the real-time supervisory control of distributed infrastructure; digital libraries?

What correlations and relationships exist between the elements of this process model and information practices? Are certain information practices common within particular sub-processes but rare in others?

Are particular sub-processes more or less likely to contribute to the unpredictability of community outcomes? For example, is time to resolution more dependent upon triage, negotiation, or collaborative debugging?

Application of Additional Data Sources and Additional Research Methods

Distributed software development communities commonly use several other digital data sources and these data sources are likely to contain information related to SWPM. These data sources include code repositories (source code version control systems), e-mail archives, discussion group / news group archives, and chat archives. Particular communities might also utilize other kinds of digital information in support of SWPM. Other non-digital information may also relate, such as hand-written notebooks, face-to-face conversations, and telephone conversations.

This project relies exclusively on the reading of a limited number of types of texts to support the analysis of distributed SWPM. Many questions have been raised that cannot be addressed using the methods currently employed. For example, many
questions have been raised about how, when, and to what effect community members make use of the inter-bug report relationships that define bug report networks. The data used in this project show that BRNs occur frequently and are extensive, but the data cannot reveal what value the new information structures created by the assertion of BRN relationships has to the SWPM process. Answering questions about the ways BRNs are utilized by the community is important to deciding whether to invest time and effort into effective automatic detection and representation of BRNs. Additional methods, like participant / observation and interviewing, should be applied to obtain access to information not accessible using the present methods.

We can also investigate the generalizability of the results presented here by examining bug reports and SWPM in other settings, such as traditional software development organizations or in other F/OSS communities with different characteristics (systems level software as opposed to end user applications; tens of participants instead of hundreds of participants; low instead of high daily bug report creation rate). We would also like to examine problem and information management practices in settings other than software development, such as real-time supervisory control of distributed infrastructure, medical diagnosis, and management of information systems.

**Application of Findings to Tool Development**

Several ideas for computational tools to improve SWPM practice have been raised during the analysis of the bug reports. These include (1) creation of a duplicate detector to support early and accurate detection of duplicate bug reports (2) a tool that completely and explicitly defines chained transitivity relationships (e.g., B is a duplicate of A; C is a duplicate of B; C must therefore be a duplicate of A, but the bug report repository studied in this project does not create the explicit relationship between C and A); (3) tools to identify, quantify, and provide representations / visualizations of bug report networks; changes to the bug report repository to make BRNs first-class information objects with attendant capabilities (e.g., support searching for and
identification of BRNs; support the association of status and resolution values to BRNs); and (4) tools to identify and mark significant but hard-to-identify information or activity, such as negotiation.

The results of this qualitative investigation may also be used to support development of computational tools for supporting research into SWPM and other kinds of distributed work. One example is a computationally based tool that can be used to recover information about design discussions and decisions from bug reports, based upon linguistic markers identified by qualitative examination of a large sample of bug reports (Ripoche & Gasser, 2003). Another example is a tool to detect linguistic patterns (e.g., instances of negotiation) that are significant predictors of community SWPM performance (e.g., identify bug reports having characteristics that can predict outcome metrics, such as long time to resolution).

**Toward a Theory and Model of Distributed SWPM**

One area of investigation largely left for future work is the development and testing of a model of SWPM. In this section, a few provisional ideas for how such a model might be constructed using the findings from this project are presented. Some of these ideas were also touched upon earlier in this and previous chapters.

One set of goals relates to one of the larger goals of our project, determining factors that lead to better or worse performance by communities involved in SWPM. Two potential dependent (albeit similar) variables are (1) whether the bug report’s trajectory results in resolution of any kind or not and (2) the elapsed time, measured in days, between the time the bug report is created and the time bug report is finally resolved.

The current project has identified several potential independent variables that might relate to these dependent variables. Negotiation is one factor identified by the current project that might correlate to SWPM performance. Having identified numerous instances of negotiation in a sample of bug reports, it is possible to test for the effects of
the presence or absence of instances of negotiation on SWPM management outcomes. Specifically, we can test for (1) the effect of the presence or absence of negotiation; (2) the effect of negotiation of specific issues; (3) the effect of the length and complexity of negotiation; (4) the effect of negotiation of multiple issues within a single bug report (e.g., negotiation about whether the bug report represents a problem or not combined with negotiation about the correct design of the patch).

The effect of membership of a bug report in a bug report network would also be interesting to test as an independent variable. Two possible BRN factors are (1) the number of other bug reports in the given network and (2) the type of the relationship of the bug under test to related bug reports.

Third, measures of the distributed dynamic cooperative ensembles associated with a particular bug report might also be a factor in SWPM outcomes. Possible factors are (1) the size of the DCE (i.e., the number of distinct actors) and (2) the set of roles represented by the DCE (see Chapter Eight).

Other factors or data from the bug reports can also be included in the relationships as well. The bug report’s severity is a likely influence on outcomes. Bug reports identified as a “blocker” (“Blocks development and/or testing work”) or “critical” (“crashes, loss of data, severe memory leak”) are most likely to get early and intense attention by the community. (Mozilla.org, *A Bug’s Life Cycle*).

Finally, the relationship of a particular bug report’s state sequence to the dependent variables is of interest. Two ways to examine this factor are (1) categorizing the state sequences by the number of transitions present in the pattern; (2) comparing the effects of particular transitions (e.g., I would expect that a state sequence that includes re-opening is likely to occur in bug reports with longer times to resolution than bug reports without re-opening); (3) the capability of the presence of negotiation (for example) to predict sub-optimal or deviant future bug report state sequences.

Certain classes of bug reports (e.g., “meta” bug report; bug report marked resolved as duplicate of another bug report) would in all likelihood need to be excluded from tests such as these. “Meta” bug reports, for example, cannot have resolutions
(software patches) in the usual sense. Similarly, marking a bug report as resolved as
duplicate of another bug report does not imply that the duplicated bug report has an
associated successful patch. Further qualitative examination of the current sample would
likely provide a clearer sense of which bug reports could be tested in this way.

One kind of model that could be investigated is a model that, based upon
information and activity present during the early phases of a bug reports lifecycle, might
predict the community’s performance managing a particular bug report. For the purpose
of creating such a model, we might define performance as a scale where shorter elapsed
time to correct problem resolution means better performance. It would be useful if this
type of model could identify activity that, if present, is likely to increase elapsed time to
correct problem resolution.
LIST OF REFERENCES


Mozilla.org *Bugzilla Keyword Descriptions List*. Available: 


http://opensource.org/licenses/osl.php; accessed [2004, April 15].


# APPENDIX A: CODE LIST

<table>
<thead>
<tr>
<th>Code</th>
<th>Code List</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Articulation - change BR assignment</td>
<td>An expression of change the BR component 'assigned to'</td>
<td></td>
</tr>
<tr>
<td>Articulation - change BR community member list</td>
<td>An expression of modifying the 'cc' list associated with a BR.</td>
<td></td>
</tr>
<tr>
<td>Articulation - change BR component</td>
<td>An expression of change the BR component 'component'</td>
<td></td>
</tr>
<tr>
<td>Articulation - change BR priority</td>
<td>An expression of change the BR component 'priority'</td>
<td></td>
</tr>
<tr>
<td>Articulation - change BR product</td>
<td>An expression of change the BR component 'product'</td>
<td></td>
</tr>
<tr>
<td>Articulation - change BR resolution</td>
<td>An expression of change to the BR component 'resolution'</td>
<td></td>
</tr>
<tr>
<td>Articulation - change BR severity</td>
<td>An expression of change the BR component 'severity'</td>
<td></td>
</tr>
<tr>
<td>Articulation - change BR status</td>
<td>An expression of change the BR component 'status'</td>
<td></td>
</tr>
<tr>
<td>Articulation - change BR target milestone</td>
<td>An expression of change the BR component 'target milestone'</td>
<td></td>
</tr>
<tr>
<td>Articulation - change other BR component</td>
<td>An expression of change to one of several BR components: use for changes to Status Whiteboard; Keywords; Status</td>
<td></td>
</tr>
<tr>
<td>Articulation - code review notice</td>
<td>An expression of a request for or completion of a code review</td>
<td></td>
</tr>
<tr>
<td>Articulation - directive</td>
<td>An expression of a directive for a particular community member, or one of N persons playing a certain role within the community, to take particular action. May be indicated explicitly or implicitly.</td>
<td></td>
</tr>
<tr>
<td>Articulation - other</td>
<td>An expression of articulation of an uncommon type. E.g., accepting a bug assignment;</td>
<td></td>
</tr>
<tr>
<td>Articulation - planning</td>
<td>An expression of a plan of action by a community member.</td>
<td></td>
</tr>
<tr>
<td>Articulation - request for favor</td>
<td>A notably polite expression for an action to be taken by another community member; contrast to Articulation - directive.</td>
<td></td>
</tr>
<tr>
<td>Attachment</td>
<td>A reference to an object attached to the BR; related email discussion, etc. Do not use for test cases or patches, which have their own specific</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Comment</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>BR component - last activity</td>
<td>The date of the last action taken on this BR. (Should this be the last bug-related action or the last action of any kind? Sometimes, 'housekeeping' activities occur long after bug-related activity ends.)</td>
<td></td>
</tr>
<tr>
<td>BR component - opened</td>
<td>The value of the BR component 'opened'</td>
<td></td>
</tr>
<tr>
<td>BR component - product</td>
<td>The value of the BR component 'product'</td>
<td></td>
</tr>
<tr>
<td>BR component - resolution</td>
<td>The value of the BR component 'resolution'</td>
<td></td>
</tr>
<tr>
<td>BR component - status</td>
<td>The value of the BR component 'status'</td>
<td></td>
</tr>
<tr>
<td>BR component - summary</td>
<td>The value of the BR component 'summary'</td>
<td></td>
</tr>
<tr>
<td>Code snippet</td>
<td>A piece of code included in the bug report.</td>
<td></td>
</tr>
<tr>
<td>Debug output</td>
<td>A reference to an object that is intended to provide details on bug conditions and behavior. Examples are stack traces, error messages, a snippet of code, etc.</td>
<td></td>
</tr>
<tr>
<td>Negotiation - bug priority</td>
<td>An expression of opinion regarding what the priority of the bug is. Don't confuse with negotiation of bug resolution scheduling.</td>
<td></td>
</tr>
<tr>
<td>Negotiation - bug resolution scheduling</td>
<td>An expression of opinion regarding when (e.g., which release) the resolution is needed.</td>
<td></td>
</tr>
<tr>
<td>Negotiation - defining a correct resolution</td>
<td>An expression of opinion regarding what defines a resolution to this bug.</td>
<td></td>
</tr>
<tr>
<td>Negotiation - feature, bug or enhancement?</td>
<td>An expression of opinion regarding whether the behavior reported as a bug is actually a bug or something else: a feature; the way the system ought to work; a 'fact of life' outside the scope of the system; a request for enhancement, etc.</td>
<td></td>
</tr>
<tr>
<td>Negotiation - has the cause been determined?</td>
<td>An expression of opinion regarding whether the bug's cause has been determined or not.</td>
<td></td>
</tr>
<tr>
<td>Negotiation - how BR should be managed</td>
<td>An expression of opinion regarding how this BR should be managed with regard to the bug. Examples: refinement; splitting off parts of the bug</td>
<td></td>
</tr>
<tr>
<td>Negotiation - identifying bug cause</td>
<td>An expression of opinion regard what the cause of the bug is</td>
<td></td>
</tr>
<tr>
<td>Negotiation - is the bug fixed or not?</td>
<td>An expression of opinion regarding whether the bug has been resolved or not.</td>
<td></td>
</tr>
</tbody>
</table>
| Negotiation - is there a problem or not? | An expression of opinion regarding whether the bug described exists or not. May be
<table>
<thead>
<tr>
<th>Code List</th>
<th>Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td></td>
<td>characterized by community member not being able to reproduce the bug or being able to reproduce the bug.</td>
</tr>
</tbody>
</table>

- **Negotiation - is this a duplicate bug report?**
  An expression of opinion regarding whether this bug report is a duplicate of another bug report.

- **Negotiation - what is the scope of the bug?**
  An expression of opinion regarding the scope of the bug, or how widespread or how it manifests in different parts of the system or under different circumstances.

- **Negotiation - what type of bug is it?**
  An expression of opinion regarding what type of bug this is.

- **Negotiation - whose responsibility is it?**
  An expression of opinion regarding who has responsibility for (that is, who should be assigned to) the bug and the bug report.

- **Patch**
  A reference to or an embedded object that represents code to resolve the bug.

- **Red herring**
  A higher level situation the can occur within a bug report. Occurs when a community member(s) proposes or pursues a problem cause and/or solution that is later refuted.

- **Related BR**
  An expression of a relationship between this bug report and another bug report (duplicate, blocks, is blocked by, etc.).

- **Statement - best practice**
  An expression of how some kind of work (e.g., object/memory allocation and release) should be done. May be done to help others learn best practice.

- **Statement - bug analysis**
  An expression of some of all of the details of the bug behavior (not the bug fix)

- **Statement - bug cause**
  An expression of the definitive or accepted cause of the bug.

- **Statement - bug description**
  An expression of how the bug manifests itself.

- **Statement - bug recreation**
  An expression of the actions to take to make the bug re-occur.

- **Statement - bug type**
  An expression or description of a "bug type"

- **Statement - BZ process issue**
  An expression of opinion regarding the s/w problem management process

- **Statement - contextual information**
  Examples include providing details on system being used when bug occurred, etc.

- **Statement - defining correct resolution**
  An expression of what defines a resolution to this bug.
<table>
<thead>
<tr>
<th>Code List</th>
<th>Code List</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement - diagnostic process</td>
<td>An expression of the process used to diagnose the cause of this bug.</td>
<td></td>
</tr>
<tr>
<td>Statement - difficulty recreating bug</td>
<td>An expression that someone is unable to create the bug.</td>
<td></td>
</tr>
<tr>
<td>Statement - fix description</td>
<td>An expression of what steps were taken to resolve this bug.</td>
<td></td>
</tr>
<tr>
<td>Statement - how to perform system action</td>
<td>An expression of how the system can do what the reporter says is a problem.</td>
<td></td>
</tr>
<tr>
<td>Statement - how to recreate</td>
<td>An expression of what steps were taken to recreate this bug.</td>
<td></td>
</tr>
<tr>
<td>Statement - hypothesis</td>
<td>An expression of a community member’s idea of what the cause of the problem is.</td>
<td></td>
</tr>
<tr>
<td>Statement - mistake</td>
<td>An expression of a mistake being noticed or corrected</td>
<td></td>
</tr>
<tr>
<td>Statement - offer of help</td>
<td>An expression of a community member to provide assistance in resolving the bug.</td>
<td></td>
</tr>
<tr>
<td>Statement - question</td>
<td>A community member poses a question to be addressed by another community member. May or may not be directed at a particular individual.</td>
<td></td>
</tr>
<tr>
<td>Statement - rationale</td>
<td>An expression of why an action is being taken, often signaled by words like '… because' or '… for'.</td>
<td></td>
</tr>
<tr>
<td>Statement - reconfirm bug</td>
<td>An expression of reconfirmation of the bug.</td>
<td></td>
</tr>
<tr>
<td>Statement - reference / pointer to code</td>
<td>An expression of where to find the relevant code</td>
<td></td>
</tr>
<tr>
<td>Statement - related experience</td>
<td>An expression of an experience relating but somewhat tangential to the bug or BR; e.g., telling a story</td>
<td></td>
</tr>
<tr>
<td>Statement - request for enhancement</td>
<td>A comment that expresses a need for further system enhancement beyond the resolution of the bug represented by the bug report.</td>
<td></td>
</tr>
<tr>
<td>Statement - request for help</td>
<td>A comment that expresses a need assistance from someone (a specific someone or more generally) in the community.</td>
<td></td>
</tr>
<tr>
<td>Statement - response to question</td>
<td>A response to a question posed in the bug report.</td>
<td></td>
</tr>
<tr>
<td>Statement - social framing</td>
<td>Examples of people adding comments that are social and non-technical in nature. It may be possible to identify a range of different types of social framing (e.g., jokes, flaming).</td>
<td></td>
</tr>
<tr>
<td>Statement - status</td>
<td>An expression by a community member of what has been done with regard to the BR and bug.</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Comment</td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Statement - temporary fix</td>
<td>An expression of a fix made in the code to work around a problem; such a fix should be replaced by a better solution in the future</td>
<td></td>
</tr>
<tr>
<td>Statement - verification</td>
<td>An expression of the act of verifying that the resolution is verified, including descriptions of the verification procedure.</td>
<td></td>
</tr>
<tr>
<td>Statement - workaround</td>
<td>An expression of actions that can be taken in lieu of a true resolution</td>
<td></td>
</tr>
<tr>
<td>Statement - workaround BR system</td>
<td>An expression of bug report activity taken in order to deal with a BR system bug.</td>
<td></td>
</tr>
<tr>
<td>Test case</td>
<td>A reference to an object, or a web site (see URL field), that can be used to recreate and verify resolution of a bug.</td>
<td></td>
</tr>
</tbody>
</table>
CURRICULUM VITAE

Robert J. Sandusky
2231 N. Monticello Ave.
Chicago, Illinois 60647
sandusky at uiuc dot edu

EDUCATION

Ph.D. Library and Information Science, University of Illinois at Urbana-Champaign.

M.S., Computer Science, Northern Illinois University, DeKalb, Illinois

B.A., English, Northern Illinois University, DeKalb, Illinois

PUBLICATIONS

Refereed Journal Papers


Refereed Conference Papers


Other Conference Papers and Presentations


Workshops


Invited participant, UIUC - NSF Workshop on Continuous (Re)Design of Open Source Software, University of Illinois at Urbana-Champaign, October 8-9, 2003.


Reports


Book Reviews


**RESEARCH**

University of Illinois at Urbana-Champaign

Research Associate, Information Systems Research Laboratory, 2002-present.

This project’s goal is the development of a new, integrative theory of situated information practices and knowledge management in large, distributed organizations. Significant findings include the role of dynamic bug report networks in the co-constitution of information, social order, and activity and the role of negotiation in shaping activity, information, and social order. The study is based upon the examination of software problems and related phenomena in particular socio-technical contexts. My work examines how free/open source software (F/OSS) development communities manage software problems by identifying and analyzing the phenomena found in F/OSS bug report repositories. Bug report repositories are fundamental and complex systems that support the continuous distributed F/OSS development process. These repositories share important features with digital libraries in their structure, complexity and relationships to work practices. Bug reports are the tangible representations of problematic system behavior discovered by the F/OSS community during testing, normal system use and the continuous (re)design process. The phenomena found in the content of bug reports from one F/OSS repository include work activities, processes, contexts, and social order. The analysis takes a qualitative, social informatics approach, placing the social aspects of knowledge structures and processes, as well as the uses and impacts of the bug report repository, in the foreground. I examine the mutually constitutive effects between these phenomena and process, context, and social order. Results from the qualitative analysis are also being applied to the design of computational tools for software problem management research and practice.
Grant: National Science Foundation ITR (Digital Society and Technologies) Program, Grant No. 0205346; Leslie G. Gasser, Principal Investigator.

Research Assistant, CANIS Laboratory, 1997-2000.
Designed and built portions of a path recording, path indexing, and path matching system for an advanced information retrieval system called the Interspace. The Interspace is a prototype environment for semantic indexing of multimedia information in a test bed of real collections, using statistical clustering of concepts and categories. The design allows users to navigate and search using the statistically generated clusters instead of term-based searches and manually constructed taxonomies. My research goal was to examine the effectiveness of making persistent representations of the activities of users (path data) available to later users and determine the effects on information retrieval effectiveness in a semantic, concept-based information retrieval system when users do and do not have access to path data.

Grant: DARPA Information Management Program, contract N66001-97-C-8535; Bruce R. Schatz, Principal Investigator.

Member of the Social Science team comprised of two faculty members and four graduate assistants. Team goals included documenting and analyzing extent and nature of system use and non-use; contributing to theoretical understanding of the changing information infrastructure and how it is transforming engineering and library work, communication, and learning practices; developing new methods for conducting user-based digital library research; and contributing to understanding of how large-scale information system design work is conceived and carried out. The team designed an integrated research program that combined broad study of use with deep study of social phenomena. Contributed to all aspects of the team’s research agenda. Particular foci included designing, building and maintaining registration and authentication system for the U of I Digital Library test bed; directing computerized data collection efforts; analyzing collected data; conducting interface and system usability tests, in situ observations and interviews. Contributed to team’s internal and external publications.

Grant: NSF/NASA/DARPA Digital Libraries Initiative under cooperative agreement NSF IRI 94-11318COOP; Bruce R. Schatz, William Misheo and Ann P. Bishop, Principal Investigators.
TEACHING

Graduate School of Library and Information Science, University of Illinois at Urbana-Champaign

Instructor for MS students with a focus on Digital Libraries. Course largely reworked from previous incarnations to focus on current and future trends in networked information systems. Student Evaluation Scores: 4.3/5.0 "Instructor's overall teaching effectiveness;" 4.3/5.0 "Overall quality of this course."

Teaching Assistant; advisor for several student group research projects. Geoffrey C. Bowker and Gregory B. Newby, instructors.

Fall 1994. LIS 370, Information Systems and Analysis.
Teaching Assistant; led two discussion sessions; created majoring of laboratory exercises; advisor for several student group research projects. Geoffrey C. Bowker and Gregory B. Newby, instructors.

North Central College, Naperville, Illinois

Adjunct Professor, Department of Computer Science, 1988-1989.
Instructor for introductory programming course using PASCAL at a private liberal arts college.

Northern Illinois University, DeKalb, Illinois

Teaching Assistant, Department of Computer Science, 1982-1983.

PROFESSIONAL SYSTEMS DEVELOPMENT EXPERIENCE

Senior Project Manager / Information Architect / Senior Web Developer, ComPsych Corp., Chicago, IL, 2000-present
Initiated and managed projects with teams of business stakeholders, visual designers, interaction and usability designers, content specialists and software developers. Introduced and advocated applying user-centered techniques to system design. Adapted industry software development practices (including Rational Unified Process (RUP) and Unified Modeling Language (UML)) for use
in a small, fast-paced development organization, including change control, requirements capture and management, quality assurance and problem management processes. Created metadata definitions and content architecture and led the effort to achieve internal consensus. Performed taxonomy planning and guided content specialists in its development and maintenance. Defined content management system requirements and performed significant portions of content management system development. Developed Web-based systems using BEA WebLogic, ATG Dynamo, ColdFusion and JRun application and personalization servers. Database systems include Oracle 8i/9i, SQLServer 7.0/2000. Web servers include IIS and Apache. Verity and Intermedia (Oracle) search engines. Operating systems include Solaris, NT, Windows 2000/XP. Programming languages used include Java; JSP; ColdFusion (4.5/5.0); JavaScript; XSL/XSLT. Content managed includes text, audio, video in HTML, XHTML, XML, PDF and other formats. The key system delivered was a dynamic digital library combining multi-media content and integrated access to multiple information and referral resources. Participated in the design and development of this digital library from its inception as a small-scale demonstration system to a larger, more robust system using n-tier components, personalization engine, Java, and XML.

Programmer, National Center for Supercomputing Applications (NCSA), University of Illinois at Urbana-Champaign, 1998-2000.
Developed WWW interfaces for access to NSF Supercomputing Usage Database and for direct researcher control of supercomputing resources. Used PERL, CGI, DBI/DBD, URI, and SQL (Oracle and Sybase) in this project.

Managed the Network Operations (19 staff operating 24 hours per day, 7 days per week, 365 days per year) and Network Engineering staff of 6 for the Federal Reserve's nationwide data communications network, Fednet. Network comprised of approximately one hundred and ninety T3, T1, fractional T1, and VSAT circuits; time division multiplexer nodes installed at approximately 50 sites; approximately 32 IBM 3745 front end processors; three large national VTAM/NetView complexes; packet switching nodes at approximately 20 locations; 1200 dial in / dial out circuits; modem management systems; matrix switch management systems; T3, T1, fractional T1, and low speed link encryption systems. Maintained network availability in excess of 99.98%. Network used for electronic transfer of funds, securities, and electronic payments and settlements. Managed many large-scale, multi-year network software upgrade projects, network hardware implementation projects, and software product development projects. Shared responsibility for multi-million dollar annual budget. I held
positions of increasing responsibility during 11-year tenure with the Federal Reserve System.

LIBRARY EXPERIENCE

   Performed copy cataloging (OCLC) with specializations in serials and materials from Southeast Asia. Participated in implementation of the library's first OPAC.

   Performed serials management and circulation desk duties as a student worker in Physics and Chemistry library.

FELLOWSHIPS, GRANTS, AWARDS


Eliza Luehm Latzer Scholarship, Graduate School of Library and Information Science, University of Illinois at Urbana-Champaign, 1995-1998.

Anita and Marie Hostetter Fellowship, Graduate School of Library and Information Science, University of Illinois at Urbana-Champaign, 1994-1995; 1998-1999.

SERVICE AND PROFESSIONAL ACTIVITIES

Referee for:

Journals:
   Computer Supported Cooperative Work: The Journal of Collaborative Computing
   The Information Society: An International Journal
   Information Systems Research
   Katherine Sharp Review

Conferences, workshops, etc.:
   First International Conference on Open Source Systems, 2005 (Program Committee)
   Mining Software Repositories Workshop, 2004
Service:
Ph.D. Student Representative to the GSLIS Ad-Hoc Committee on Technology Courses, 1995-1996.
Ph.D. Student Representative to the GSLIS Faculty Committee, 1994-1995.

Affiliations:
Member Association for Computing Machinery (ACM)
Member American Society for Information Science and Technology (ASIST)
Member Association for Library and Information Science Education (ALISE)

SKILLS

Design and Execution of Qualitative and Quantitative Research Projects Using:
  Interviewing; observation; surveys/questionnaires; transaction/system log analysis; usability studies

Analysis of Qualitative and Quantitative Data:
  Grounded theory; content analysis; statistical analysis

Computational Tools for Qualitative and Quantitative Data Analysis:
  Atlas.ti; SPSS; SAS and others

Systems Analysis and Development Skills:
  Software Development Methods: Rational Unified Method (RUP); Unified Modeling Language (UML); Design Patterns; Structured Analysis and Design.
  Programming Languages and Systems: Java; BEA WebLogic 8.1; ColdFusion 4.5/5.0 Application Server; ATG Dynamo Application / Personalization / Scenario Servers 5.6/6.2; Microsoft SharePoint Portal Server; JRun4 Application Server; JavaScript; PERL/CGI; Smalltalk; HTML / XHTML / XML / XSL / XSLT; SQL (Oracle, Sybase, SQLServer, Access); PASCAL; Basic Assembly Language; COBOL; BASIC; C. Databases: Oracle; SQLServer; Sybase; Access; Versant (OODBMS); IMS, DB/2 (Hierarchical). Operating Systems / Protocols: Unix (Linux; Solaris; HP Unix); Windows (XP; 2000; NT; 98; etc.); MVS; VM. Protocols: Internet; OSI; SNA

Management:
  Project Management; Software Development, Deployment and Maintenance; Systems (Hardware and Software) Deployment and Maintenance; Personnel Management; Budget Planning and Management

Foreign Language:
  German
REFERENCES

References are available upon request.