

ULRR

Open source programmers' information seeking

Item Type	Thesis
Authors	Sharif, Khaironi Yatim
Download date	2026-06-13 02:23:55
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/2853



UNIVERSITY *of* LIMERICK

OLLSCOIL LUIMNIGH

Open Source Programmers' Information Seeking

**Submitted by Khaironi Yatim Sharif
For the award of Doctor of Philosophy**

**Supervised by
Dr. Jim Buckley**

**Submitted to the University of Limerick
June 2012**

Abstract

Several authors have proposed information seeking as an important perspective for the study of software maintenance and evolution, and have characterized information seeking empirically in commercial software evolution settings. However, there is little research reported in the literature that describes the information seeking behavior of open source (OS) programmers, even though OS contexts would seem to exacerbate information seeking problems for maintenance programmers. In the OS setting, team members are typically delocalized from each other and are more likely to use asynchronous communication.

In addition, the information seeking behavior of OS system maintainers is an increasingly important research topic, due to the increasing body of OS software needing to be maintained and users' increasing reliance on OS software systems. Such an increased reliance and prevalence means that there is increased demand for software maintenance of OS systems and it is this area that this thesis will strive to address.

This thesis reports on a series of empirical studies that classify OS programmers' information needs during maintenance. In these studies, types of information needs are identified through a Grounded Theory based analysis of the questions that appear in their developer mailing lists. Then the prevalence of these information needs, on different mailing lists, are quantified and the responses obtained are characterized. In doing so, several interesting observations are made about the types of information these programmers seek, the likelihood that they will receive responses and the number of responses they are likely to get.

The findings of this thesis largely reflect the academic articles in this area, documenting programmer's focus on implementation specific detail and team awareness. However interesting additional insights were gained with respect to the importance of the technology around software evolution in an OS context, and the programmers' design and documentation information needs. Also, a surprisingly low rate of response from the OS community to mailing list requests was noted.

Acknowledgements

Firstly, I thank my wife, Azah, and my children, Amira, Aiman and Amar, for their never-ending support and sacrifices. I wish to also acknowledge my mother, Moktiaton, and my late father, Sharif, for introducing me to life and knowledge.

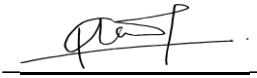
I am extremely grateful to my supervisor, Dr Jim Buckley, for his guidance and wise words. In particular, I thank Dr Buckley for his commitment to research and teaching that inspired me throughout the undertaking of the thesis.

My thanks go to the Malaysian Government for providing a full sponsorship for my PhD research. My study was also supported, in part, by a Science Foundation Ireland grant (03/CE2/I3031) to attend the Irish Software Engineering Research Centre (Lero). I am grateful to the Irish Software Engineering Research Centre for this funding and for the other forms of support to finish my PhD.

Finally, I extend my thanks to my colleagues and the university staff. Everyone within the Department of Computer Science and Information Systems (CSIS) was willing to assist, and provided me with great advice, encouragement and motivation.

Declaration

The work described in this thesis is, except where otherwise stated, entirely that of the author and has not been submitted in any part for a degree at this or any other University.

Signed:  _____

Khaironi Yatim Sharif

Date : 18th June 2012

Table of Contents

CHAPTER 1	Introduction	2
1.1	Research Motivation	2
1.2	Program Comprehension in Software Maintenance	4
	Information Seeking as a Core Element in Program Comprehension	4
1.3	The Importance of Open Source	6
1.3.1	The Prevalence of Open Source Software	6
1.3.2	Software Maintenance in Open Source Software	9
1.4	Thesis Focus	10
1.5	Research Questions	11
1.6	Contributions	12
1.7	Layout of Thesis	13
1.8	Main Researcher’s Background	14
CHAPTER 2	Program Comprehension and Information Seeking	17
2.1	Introduction	17
2.2	What is Program Comprehension?	17
2.3	Program Comprehension Theories	18
2.3.1	Top-Down and Bottom-Up Comprehension	18
2.3.2	Encompassing Theories	20
2.4	Information Seeking and Software Comprehension	25
2.4.1	What is Information Seeking?	26
2.4.2	Information Seeking Models	27
2.4.3	Information Seeking in Software Maintenance	31
2.5	Information Seeking-Related Issues for Maintenance Programmers	33
2.5.1	Information Sources	34
2.5.2	Information Sought	36
2.5.3	Concept / Feature Location	41
2.6	Information Seeking in Open Source Software Development	43

2.6.1	Open Source Definition.....	44
2.6.2	Open Source Software Development Process.....	46
2.6.3	OS Programmers' Communication Channels	50
2.6.4	Categorizing Open Source Projects.....	52
2.7	Locating this work with other research in the field	55
2.7.1	Differences between This Study and Previous Works	56
CHAPTER 3 Methodology and Implementation.....		59
3.1	Introduction.....	59
3.2	Qualitative and Quantitative Approaches.....	59
3.2.1	Quantitative Research	59
3.2.2	Qualitative Research.....	61
3.2.3	Comparison between Qualitative and Quantitative Methods	62
3.3	Data Analysis Methods	66
3.3.1	Grounded Theory	66
3.3.2	Content Analysis	72
3.3.3	Other Methods of Textual Analysis	75
3.4	Method of Data Analysis Employed	75
3.5	Sample Size and Validity	78
3.6	Method of Data Collection	79
3.7	Research Ethic	80
3.8	Coding Process and Analysis Schema Development.....	80
3.9	Conclusion.....	82
CHAPTER 4 Schema Development.....		84
4.1	Introduction.....	84
4.2	The Analysis Process	84
4.3	Pilot study on the Initial Dataset (1st Observation-cycle).....	86
4.3.1	The Dataset	87
4.3.2	Schema Evolution	88
4.3.3	The Information Seeking Schema.....	93

4.3.4	Mapping the provisional information type into the new schema	93
4.3.5	Information Focus	95
4.3.6	Information Aspect	97
4.3.7	Knowledge Strength	99
4.3.8	Schema application result and further refinement	100
4.4	Observation on Different Stages of Software Evolution	102
4.4.1	The Dataset	103
4.4.2	Data Analysis and Schema Refinement	103
4.5	Stress Testing the Schema with a Large Dataset	110
4.5.1	Method and the Dataset	111
4.5.2	Results and Schema Refinement	112
4.6	Observation on Various Categories of OSS Projects	115
4.6.1	The Dataset	115
4.6.2	Results and Schema Refinement	119
4.7	Conclusion	122
CHAPTER 5 Schema Application		124
5.1	Introduction	124
5.2	Empirical Study	124
5.2.1	Objectives	124
5.2.2	Dataset	124
5.2.1	Protocol	126
5.3	Analysis Result	127
5.4	Result Discussion	129
5.4.1	Information Focus	129
5.4.2	Information Aspect	133
5.4.3	Information Focus & Aspect	137
5.4.4	Trend over OS Project Category	142
5.5	Responses to Information Request	146
5.5.1	Low Response Rate	148
5.5.2	High Rated Information Types	154

5.5.3	Discussion on Answered Questions.....	154
5.5.4	Response Time-span.....	155
5.6	Conclusion.....	156
CHAPTER 6 Conclusion and Future Works		159
6.1	Introduction.....	159
6.2	Contributions	159
6.3	Revisiting the Research Questions.....	160
6.3.1	Information artifacts probed by programmers	160
6.3.2	Information sought within the probed artifact	161
6.3.3	Clusters of Information Artifacts and their Information Sought .	162
6.3.4	Responses gained from the open source programmers' information seeking activities.....	162
6.4	Implications of this work.....	163
6.5	Limitation of this research study	165
6.6	Future work.....	167
6.7	Conclusion.....	169

List of Tables

Table 2.1 - OS project categorization by Feller and Fitzgerald (2002)	52
Table 2.2 - OS project categorization by Daniel et al. (2009a).....	54
Table 3.1- Summary of main features of qualitative and quantitative research, derived from Neill (2007).....	64
Table 3.2 - Quantitative and qualitative studies in ESP	65
Table 3.3 - Quantitative and qualitative studies in ESE	65
Table 4.1- Dataset for pilot study.	87
Table 4.2 - The provisional information types identified during pilot study	89
Table 4.3 - Refined of initial information type.....	90
Table 4.4 - The coverage provided by the Information Seeking Schema.....	93
Table 4.5 - Information Focus	96
Table 4.6 - Information Aspect.....	97
Table 4.7- Knowledge Strength	99
Table 4.8 - Information Focus of BSF and JDT.....	100
Table 4.9 - Information Aspect BSF and JDT	100
Table 4.10 - Knowledge Strength of BSF and JDT	101
Table 4.11 - Dataset for Second Observation-cycle.....	103
Table 4.12 - New Categories of questions in bucket.....	104
Table 4.13 - Content Analysis Result for Information Focus Dimension (2nd Observation-cycle).....	105
Table 4.14 - Content Analysis Result for Information Aspect Dimension	
(2nd Observation-cycle).....	106
Table 4.15 - Finer Grained Categories of System Implementation category.....	107
Table 4.16 - Finer Grained Categories of Tools/Technology category.....	108
Table 4.17 – Dataset for 3rd Observation-cycle.....	111
Table 4.18 - Content Analysis Result for Information Focus Dimension (3rd Observation-cycle).....	112
Table 4.19 - Content Analysis Result for Information Aspect Dimension	113
(3rd Observation-cycle).....	113

Table 4.20 - Datasets used in 4th Observation-cycle.....	116
Table 4.21 - Category for each mailing list used in 4th Observation.	118
Table 4.22 - The schema applicability on Feller and Fitzgerald's (2002) Categories.....	118
Table 4.23 - Content Analysis Result for Information Focus Dimension (4th Observation-cycle).....	119
Table 4.24 - Content Analysis Result for Information Aspect Dimension (4th Observation-cycle).....	119
Table 4.25 - Low requested Information Focus.....	120
Table 5.1 - Description for OSS projects used in this thesis.	125
Table 5.2 - All dataset for this thesis.	125
Table 5.3 - Reanalysis Result for Information Focus Categories	127
Table 5.4 - Reanalysis Result for Information Aspect Categories.....	127
Table 5.5 - Response Analysis Result for Information Focus and Aspect Dimension.....	128
Table 5.6 - Process Related Issues of Information Focus.....	136
Table 5.7- Process Related Issues of Information Aspect.	136
Table 5.8 - Relationship between Information Focus Categories and Information Aspect Categories.....	138
Table 5.5 - Response Analysis Result for Information Focus and Aspect Dimension(replicated from earlier)	148
Table 5.9 - Response Analysis Result for Tools & Technology Based Information Focus.	150

List of Figures

Figure 1.1 - Distribution of maintenance activities by categories suggested by (Swanson, 1976).....	3
Figure 2.1 - Meta-model of Von Mayrhauser and Vans (1995)	24
Figure 2.2 - Information seeking in context (Niedźwiedzka, 2003).....	27
Figure 2.3 - Correlation of information seeking theories, adapted from O'Brien (2007).....	29
Figure 2.4 - ISM for software maintenance (O'Brien, 2007).....	31
Figure 2.5 - Concept location process (Chen and Rajlich, 2000)	42
Figure 2.6: The open source definition by Open Source Initiative (2009)	45
Figure 2.7- Information seeking in OS software maintenance	55
Figure 3.1 - The Grounded Theory “data dance” (Kelsey, 2003)	67
Figure 3.2 - Grounded Theory flowchart (Bitsch, 2005)	69
Figure 3.3 - The Grounded Theory analytic process, adapted from Harwood (2002)	70
Figure 3.4 - Open coding sequence according to Strauss and Corbin (1998)	71
Figure 3.5 - Seven-phase model for data collection and analysis	76
Figure 3.6 - Process of category creation	82
Figure 4.1 - Schema for open source programmer’s information seeking.....	92
Figure 4.2 - Finalized Schema for open source programmer’s information seeking.....	121
Figure 5.1 - Information Focus for all OS project in the dataset.	129
Figure 5.2 - Percentage of request for IDE focus in each individual OS project.	132
Figure 5.3 - Information Aspect for all dataset for this thesis.	133
Figure 5.4 - Clusters of Information Focus and Information Aspect.	139
Figure 5.5 - Percentage of Request for implementation centric focus over all OS project.	143
Figure 5.6 - Request for System Documentation Focus over all OS projects. ..	145

Figure 5.7 - Information Aspects of System Bug.....	149
Figure 5.8 - Information Aspects of Support Required.....	150
Figure 5.9 – Info. Aspects of Communication Channel Info. Focus	151
Figure 5.10 – Info. Aspects of Documentation Info. Focus	152
Figure 5.11 - Information Focus Categories for Awareness Aspect.....	153

List of Appendices

Figure A1 – Personal Email Communication With One Author Of Daniel et al (2009).	189
Table A1 - Information Focus Updated During 2nd Observation.	190
Table A2 – Information Aspect Updated During 2nd Observation	192
Table B1 - Information Focus Updated During 3rd Observation.	195
Table B2 – Information Aspect Updated During 3rd Observation	197

List of PhD Work Publications

Sharif, K. Y. and Buckley, J. (2008) Developing Schema for Open Source Programmers' Information-Seeking. *In: International Symposium on Information Technology 2008 (ITSIM '08)*, September 2008 Kuala Lumpur , Malaysia: IEEE. Computer Society. 551-559

Sharif, K. Y. and Buckley, J. (2008) Observing Open Source Programmers' Information Seeking. The 20th Annual Psychology of Programming Interest Group Conference (PPIG 2008). Lancaster University, Lancaster, United Kingdom. 15 -24

Sharif, K. Y. and Buckley, J. (2009) Observation of Open Source programmers' information seeking. ICPC '09. IEEE 17th International Conference on. Program Comprehension, 2009. Vancouver, British Columbia, Canada, IEEE Computer Society. 307 – 308

Sharif, K. Y. and Buckley, J. (2009) Further Observation of Open Source Programmers' Information Seeking. The 21st Annual Psychology of Programming Interest Group Conference. Psychology of Programming Interest Group (PPIG 2009). Limerick, Ireland. 162 - 173

List of Abbreviations

API	:	Application Programming Interface
ATM	:	Automated Teller Machine
AWT	:	Abstract Window Toolkit
BSF	:	Java Bean Scripting Framework
DOD	:	Department of Defence
ECS	:	The Element Construction Set
ESE	:	International Journal of Empirical Software Engineering
ESP	:	Empirical Studies of Programmers
GUI	:	Graphical User Interface
IDE	:	Integrated Development Environment
IEEE	:	Institute of Electrical and Electronics Engineers
IP	:	Internet Protocol
IS	:	Information Seeking
ISM	:	Information Seeking Model
JDT	:	Java Development Tool
MIT	:	Massachusetts Institute of Technology
OS	:	Open Source
OS	:	Open Source Software
OSD	:	Open Source Document
SIP	:	Session Initiation Protocol

Chapter 1

Introduction

CHAPTER 1 Introduction

1.1 Research Motivation

Software maintenance has been defined by the Institute of Electrical and Electronics Engineers (IEEE) (1998) as the “modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment”. These software maintenance activities make software systems evolve and change over time. In this context, Belady et al. (1976) define software evolution as “the dynamic behavior of programming systems as they are maintained and enhanced over their lifetimes.”

Thus, software maintenance is not just “fixing mistakes” (O’Brien, 2007) – in fact, the total maintenance effort that is spent debugging (or fixing mistakes) in a system is estimated at only 20% of total maintenance effort, while the remaining 80% is spent on other maintenance activities as implied by the IEEE definition above (Pressman, 2000). Lientz et al. (1978) specified three distinct categories of software maintenance, in line with the IEEE definition:

- Corrective maintenance – corrective maintenance is carried out to respond to system failures (Swanson, 1976). It requires changes to be made to fix defects uncovered in the system.
- Adaptive maintenance – adaptive maintenance is carried out to reflect changes in the data and the processing environment (Swanson, 1976). This type of maintenance requires modification or adaptation of the software to accommodate changes to its external environment (for example moving to a new database).
- Perfective maintenance – perfective maintenance is carried out to remove inefficiencies, improve performance or enhance its application domain relevance (Swanson, 1976). This type of maintenance is required as a response to requests to enhance the functionality or the efficiency of

particular software. In other words, this type of maintenance will probably expand the software, making it exceed its original functional requirements.

In a more recent definition, the IEEE (1991) proposed another category of maintenance, namely “preventative maintenance”, to describe proactive efforts to ensure the software is more correctable, more adaptable and can be enhanced more easily in the future.

Swanson (1976) reported on the effort spent on each of the original categories of maintenance. As shown in Figure 1.1, corrective and adaptive maintenance, combined, contribute 35% of all maintenance activities, whereas perfective maintenance activities are 61% of the maintenance activities. The remaining 4% is spent on other maintenance activities that could possibly be identified as preventative maintenance.

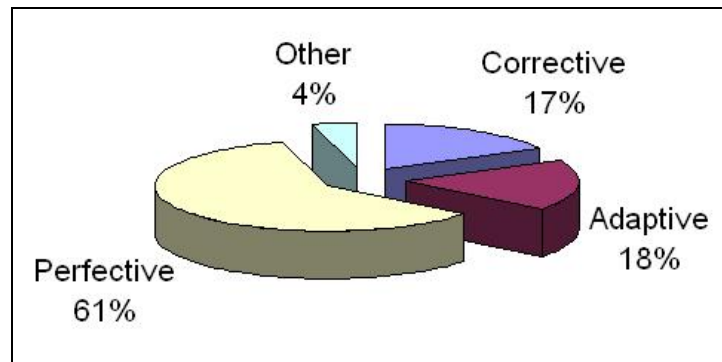


Figure 1.1 - Distribution of maintenance activities by categories (Swanson, 1976)

The finding suggests that the requirements for a successful software system will expand substantially over its lifetime, and this will require many changes to the software to accommodate the changing needs.

Consequently, software maintenance and evolution demand a large amount of the effort consumed by the software system’s lifecycle. Researchers estimate that 60% to 80% of the whole software lifecycle effort is spent on these phases (Lientz et al., 1978, Von Mayrhauser, 1993, Pressman, 2000, Zayour and Lethbridge, 2001). For example, Sommerville (2004) suggests that maintenance consumes more than 75% of the software budget. Sommerville (2004) also

suggests that 75% of software engineers spend more effort on software maintenance than any other activities in the software system lifecycle. While Kemerer and Slaughter (1999) do acknowledge that the empirical evidence for such assertions is outdated and should be reassessed, the higher complexity and scale of newer systems (Daniel et al., 2009a, Sommerville, 2004, Pressman, 2000) imply that the effort required for software maintenance will only have increased in the time since this evidence was found.

1.2 Program Comprehension in Software Maintenance

Software maintenance consists of two general stages: “understanding the program” and “actually performing the change” (Prechelt et al., 1998). The understanding stage is likely to be the biggest part in software maintenance: Programmers’ efforts to comprehend a program before (and during) successful modification are estimated to be between 50% and 90% of the overall maintenance effort (De Lucia, 1996).

Information Seeking as a Core Element in Program Comprehension

Kingrey (2002) defines information seeking (IS) as the search, recognition, retrieval and application of meaningful content. O’Brien (2005) states that such information seeking may be performed with a “specific strategy or serendipity”, resulting in the acceptance or rejection of the discovered information, and the process may proceed to a “logical conclusion or (be) aborted midstream”.

In the context of software maintenance, information seeking is recognized as a highly important sub-task in software comprehension (Curtis, 1988, Seaman, 2002, Singer, 1998, Singer and Lethbridge, 1998, Sim, 1998, O’Brien, 2007). Sim (1998), for example, refers to maintenance programmers as task-oriented information seekers, focusing specifically on getting the answers they need to complete a task using a variety of information sources. Likewise, Singer et al. (1997), in their case study of programmers’ maintenance activities in the

telecommunications domain, found that programmers perform more searching (i.e., grep-based navigation) than any other activities. More recent studies (Cleary et al., 2008, Cleary and Exton, 2007) suggest that information seeking is a very credible model for describing the goal-orientated, opportunistic software comprehension strategies employed by software engineers.

Understanding software is not necessarily restricted to extracting information from the source code only (O'Brien et al., 2001, Seaman, 2002). Various artifacts – for example any form of system documentation – can be used when extracting the required information. However, there is a possibility that non-source code documentation will be incomplete, inaccurate, missing or may have never existed (Lethbridge et al., 2003). Given these inadequacies, it is unsurprising that programmers consider the source code as their main reference point (Seaman, 2002), although she also suggests that these other information sources are trusted, colleague programmers, CASE tools and, to a lesser degree, documentation.

Software maintainers are not only required to recover information on the operation and structure of the software system, which may be explicitly represented in source code; They must also look for implicit information such as software function, design issues and the assumptions made during implementation (Yang, 1997). Given the typical scale of software systems, the software complexity and the intricacy of programming languages, these recovery tasks (information seeking) are clearly complex tasks.

1.3 The Importance of Open Source

The Open Source (OS) concept started in the 1970s (Bretthauer, 2002), when a programmer at the Artificial Intelligence Lab, MIT, named Richard Stallman asked for the driver source code to improve Xerox's laser printers. He was refused. The rejection caused him to adopt a fundamental belief that software should be free. Stallman later resigned from MIT and created the GNU operating system – a suite of free software products to demonstrate his belief. In October 1985, Stallman formed the Free Software Foundation to support the development of GNU.

The basic requirement for a software development project to be considered open source is the open availability of its source code (Feller and Fitzgerald, 2002). Bonaccorsi et al. (2003) state that OS is a process innovation: that is, it is a new and revolutionary process of producing software, based on unconstrained access to source code as opposed to the traditional closed and proprietary-based approach of the commercial world.

1.3.1 The Prevalence of Open Source Software

Since these beginnings, OS has become prevalent in the software industry (Feller and Fitzgerald, 2002). Today, OS software is well known for its high degree of reliability, efficiency, robustness and portability (Feller and Fitzgerald, 2002, Bonaccorsi and Rossi, 2003, Fitzgerald, 2004). Below are examples of OS software that are very well-known and that have gained popularity in both development and usage contexts. Some of these have shown their reliability in safety critical contexts:

i) The Linux Operating System

The Linux operating system was created by Linus Torvalds in 1991. The number of developers collaborating on the Linux operating system was estimated to exceed 40,000 (Raymond, 2001a) and the number of Linux

users was estimated at 25 million (Torvalds and Diamond, 2001). Torvalds (2001) claimed that Linux represents the largest collaborative project in history. Linux might also be one of the most prevalent pieces of OS software worldwide. In October 2009, 7 out of the 10 most reliable hosting company sites were using Linux as their web server operating system with Linux being the first ranked (Netcraft, 2009a). Illustrating this point, in October 2009, the US White House launched a new version of its website using Linux and the Drupal (another OS piece of software) content management system (Netcraft, 2009c). Likewise, Netcraft (2010) reported that Microsoft, the ubiquitous proprietary software company that produces its own web servers, also uses the Linux server for at least two of their web domains (search.microsoft.com and download.microsoft.com).

ii) Apache HTTP Server

The development of the Apache HTTP server began in February 1995 by a group of volunteers who decided to pool their expertise (Fielding, 1999). The first version of Apache was released in December 1995 and it quickly became the most widely used web server product (Bretthauer, 2002). In 2005, the Apache HTTP server had an estimated user base of over 10 million (Koponen and Hotti, 2005). In a more recent estimation, the Apache HTTP server is said to have over 40 million users and is used in over 70 percent of web servers (Netcraft, 2009b).

iii) Mozilla

Mozilla is an internet browser which was created based on the source code of Netscape's browser. In January 1998, influenced by Raymond's (2001a) classic paper, "The Cathedral and the Bazaar" which discussed the open source concept, Netscape released their source code for their browser to the public. Netscape formed a group called the Mozilla

Organization to act as the central coordination mechanism for work on the project. Mozilla has since become one of the most popular internet browsers. According to Dotzler (Dotzler, 2009), Mozilla's director of community development, Mozilla Firefox has approximately 270 million users.

iv) Sahana

Sahana is an open source Disaster Management System (Group, 2004). This is a web-based collaboration tool that addresses the common coordination problems during a disaster, from finding missing people, and managing aid to managing human resources effectively between government groups, the civil society and the victims themselves. The Sahana project was initiated by volunteers in the Sri Lankan Free and Open Source Software (FOSS) development community to help victims during the Asian Tsunami in December 2004. The system was officially used by the Government of Sri Lanka and the system was released as free and OS software.

v) Various US Department of Defence systems

A recent study by the US Defence Information Systems Agency found that the Department of Defence (DOD) currently uses a significant variety of open source computer software, and concluded that open source software is vital to DOD information security (Wilson, 2007). One of the examples of OS software used by the DOD is in Shadow 200, a tactical unmanned aircraft system. Linux is used in its ground control station to control this unmanned aerial vehicle (Gasperson, 2006, International, 2009).

These examples show that the source code sharing concepts seem to invite extensive collaboration from programmers around the world. Interestingly, they

also show that this “bazaar” development style (Raymond, 2001b) manages to produce high quality software that users rely on, even for critical safety systems. Like all successful software systems, these software systems need to be evolved and maintained.

1.3.2 Software Maintenance in Open Source Software

Given the increased prevalence of OS software and the widespread reliance on it, it is likely that software maintenance is a vital concern in this paradigm. As discussed above, information seeking underpins this concern. Hence, the study of software maintenance – and in particular, the study of information seeking – among OS programmers is a worthwhile field of endeavour that could be significant in improving not only OS software maintenance but also improving software maintenance in general.

Notwithstanding, several characteristics of the generic OS software development process have been suggested (Feller and Fitzgerald, 2002, Gacek and Arief, 2004) that might impact on the software maintenance process for OS specifically¹:

- Large, globally distributed development communities
- Truly independent peer review
- Prompt feedback on user and developer contribution
- Highly talented, highly motivated developers
- Actively involved users
- Rapid release schedules.

Some of these characteristics may impact substantially on the OS programmers' information seeking. For example, the OS software development process

¹ Details about these characteristics are discussed in Chapter 2.

generally involves (or has the potential to involve) large, globally distributed communities of developers collaborating primarily through the internet (Feller and Fitzgerald, 2002, Fitzgerald, 2004). This typical widely distributed, asynchronous development would seem to make information seeking more difficult (Sharif and Buckley, 2009a, Gutwin et al., 2004) when programmers move beyond the code base. Likewise, the availability of highly motivated and involved developers suggests that other programmers in the group will be supportive of information seeking needs and that the blocking reported in other work (Ko et al., 2007) will not be so in evidence in this work.

1.4 Thesis Focus

Our review of the literature suggests that OS software development has characteristics that exacerbate information seeking problems (Sharif and Buckley, 2009a, Gutwin et al., 2004), as team members are typically delocalized from the other members of their team and often forced into asynchronous communication. However, information seeking is recognized as a core task in software maintenance, and in turn as a core sub-task of the software development lifecycle. Given the increased prevalence of OS software, these information seeking problems could be a challenge for OS software maintenance, and thus should be addressed in the literature.

To date, there is little research on information seeking among OS programmers (Sharif and Buckley, 2009b, Sharif and Buckley, 2009a). Most studies in the information seeking field have been done in commercial or non-OS software environments. While those studies provide concrete insights into commercial and/or localized software development work (Ko et al., 2007), we still have little evidence regarding the information that maintainers look for in a distributed OS development, and whether they are successful in obtaining it. Hence, this research focuses on the study of information seeking behaviour among OS programmers.

1.5 Research Questions

This research aims to characterize information seeking in OS software projects. Thus it directs itself at the following questions:

1. What information artifacts are probed by programmers.
2. What information is sought within the probed artifacts.
3. Are there particular types of information that these programmers seek that they have difficulty in getting response through developer mailing list?

In this context, an information artifact can be described as the entity that the programmers seek information about; it is the programmers' information focus or the target entity of their information seeking attention. It should be noted that this does not refer to the medium through which the programmers seek this information. For example, when a programmer asks another programmer about the meaning of a specific piece of source code, the medium is the programmer they ask. The information artifact is the piece of source code. Thus, the information artifact reports on the physical representation of the system that the programmer is trying to find information about, but needs help doing so. This suggests that the artifact is obscure in some way in its presentation of the desired information.

The information sought within the probed artifact refers specifically to the information that the programmer can't deduce from the information artifact. Thus, research on this aspect of OS programmers' behaviour will help identify the types of information that are difficult to derive from the studied information artifacts.

Both these perspectives are the foundations of a schema that informs on the information seeking of OS programmers. In determining the information that they frequently seek but have difficulty in obtaining from specific artifacts, this research has the potential to define the requirements for visualization tools that truly support programmers in their maintenance information seeking endeavours (Buckley, 2009). Alternatively, when it is shown that programmers habitually seek specific information from inappropriate artifacts, this research has the potential to

present guidelines for programmers, showing where the information can be more easily obtained.

It is also important to investigate the responses obtained by OS programmers to their information requests. This provides an insight into the community's willingness to help other programmers. OS projects have been characterized as communities with highly proactive and motivated contributors (Feller and Fitzgerald, 2002, Gacek and Arief, 2004). At the same time, software maintenance is portrayed as an undesirable task that programmers tend to shy away from (Sommerville, 2004). This thesis looks at these two opposing perspectives and evaluates whether the pro-activity associated with OS development overcomes the reticence of programmers with regard to maintenance. If such reticence is not overcome, the need for supporting tools directed at programmers' information needs is heightened.

1.6 Contributions

The main contribution of this thesis is a derived schema of information seeking artifacts and information types in OS software development. This proposed schema is an open schema allowing further evaluation and refinement, which can then be employed in future research in this area.

The schema can deliver the following additional contributions:

- i. Identification of the information sought by programmers during their maintenance tasks.
- ii. Assessment of the current related theory with respect to real practice in OS software development. Related theories include program comprehension, information seeking/blocking, concept location and software visualization.
- iii. Identification of the areas of information insufficiency in software artifacts and software community communication, thus highlighting future work for the software visualization community.

- iv. An understanding of the strategies that programmers use to do maintenance tasks. This is important, to identify the challenges they face and to identify the corresponding opportunities for tools or processes that support the maintenance strategy as suggested by Thomas and Brad (2010).

This thesis focuses on the first three contributions, and illustrates the potential of the schema in all of these endeavours.

1.7 Layout of Thesis

The remainder of this thesis is organized as follows:

- **Chapter 2: Program Comprehension and Information Seeking.** This chapter presents a discussion of information seeking as a significant perspective in the study of software maintenance. Specifically, it discusses program comprehension as a core component of software maintenance and the important role of information seeking during program comprehension.
- **Chapter 3: Methodology and Implementation.** In order to study the information seeking behaviour of OS programmers during software maintenance, this chapter reviews the potential empirical methods that can be used. It discusses a mixed qualitative and quantitative method as the best approach to study programmers' information seeking behaviour.
- **Chapter 4: Schema Development.** This chapter describes the derivation of the information seeking schema. It discusses the application of the open coding procedure² according to the Grounded Theory approach during the creation of the initial schema and the application of the content analysis procedure during the refinement of the schema.

² In the context of Grounded Theory, coding procedure is categorizing procedure to identify concept that derived from the data.

- **Chapter 5: Schema Applications.** This chapter discusses the application of the resultant schema to reanalyse the datasets used in the study. The schema is employed to carry out content analysis on all the datasets. The analysis is done based on the frequency of request and the rates of response to frequent needs.
- **Chapter 6: Conclusion and Future Works.** This chapter presents the contributions and findings of this research resulting from the thesis questions posed in Chapter 1. Consideration is then given to possible directions for future work.

1.8 Main Researcher's Background

Although this research is distinctly contextualised within the domain of computing and software engineering, it can also be considered multidisciplinary, drawing heavily as it does upon the social sciences. An important term that is sometimes used within the social sciences is that of "Reflexivity". This broadly means that it is important to consider the role of the researcher and how their knowledge and experience may affect the final results (Holland, 1999).

In the context of this study, it is possible that the main researcher's experience related to programming and software development may have impacted upon this study and so this experience is detailed here.

The main researcher has previously worked as a programmer, consultant and also as a software engineering lecturer in a university in Malaysia. He has various programming experience including database programming with SQL (ORACLE), C and C++ programming in the development of an on-board automobile system (real-time software to support cars' artificial intelligent features) and webportal development using various internet programming language such as Javascript, JSP and PHP. He has no commercial experience in the Java language.

But he has had teaching experience of Java, Assembly language, Web Application Development, Software Design and Computer Organization. He was also a consultant in designing a web-based instructional system to be used in the university that he teaching for.

Another factor that might impact this study is main researcher's language Malaysian background. The main researcher is a non-English native speaker. This might impact this work, especially as it entails undertaking textual data in English. Both issues (programming experience and language) are discussed in section3.8.

Chapter 2

**Program
Comprehension and
Information Seeking**

CHAPTER 2 Program Comprehension and Information Seeking

2.1 Introduction

Rajlich and Wilde (2002) stated that program comprehension is an essential part of software evolution and software maintenance. According to them, “software that is not comprehended cannot be changed”. The fields of software documentation, visualization, and program design, are in part driven by the need for program comprehension (Rajlich and Wilde, 2002). Information seeking has been recognized as a core sub-task in program comprehension (Curtis, 1988, Seaman, 2002, Singer, 1998, Singer and Lethbridge, 1998, Sim, 1998, O’Brien, 2007). This chapter discusses information seeking as a significant perspective in the study of software maintenance. Specifically, it discusses program comprehension as a core component of software maintenance, and the important role of information seeking during program comprehension.

2.2 What is Program Comprehension?

Muller (1994) relates program comprehension to mental model development at different levels of software abstraction. He defines software comprehension as “the task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself, to models of the underlying application domain, for maintenance, evolution, and reengineering purposes”. In contrast to Muller, Good (1999) concentrates more on the information needed by programmers, stating that software comprehension is “a process in which the programmer uses prior knowledge about programming and information present in the program to form a dynamic, evolving model of the program, which can then be applied to the task”. In later work, O’Brien (2004) enhances this definition by expanding on the knowledge used in achieving comprehension. He defines software comprehension as “a process whereby a

software practitioner understands a software artifact using both knowledge of the domain and/or semantic and syntactic knowledge, to build a mental model of its relation to the situation”.

2.3 Program Comprehension Theories

Program comprehension theories generally centre around top-down and bottom-up comprehension (Rajlich and Wilde, 2002). Evidence suggests that bottom-up comprehension represents the comprehension technique used by a programmer who is unfamiliar with a particular application domain. Meanwhile, top-down comprehension represents the situation where a programmer is familiar with the application domain (Shaft and Vessey, 1995, O'Brien et al., 2004).

2.3.1 Top-Down and Bottom-Up Comprehension

The top-down / problem domain-orientated process of software comprehension was first proposed by Brooks (1977). He described software comprehension as recreating knowledge about the program domain and mapping it to the source code (Storey, 2005). At its core, this involves a hypothesis verification (or rejection) process.

The programmer will firstly create an initial set of hypotheses about the program domain. This hypothesis set is then refined in hierarchical order, producing sub-hypotheses. Brooks (1983) suggests that these sub-hypotheses are then verified or not depending on the presence or absence of “beacons” in the code. Brooks defines beacons as “recognizable/familiar features in the code that act as cues to the presence of certain structures or operations”. However, the definition later proposed by Harris and Cilliers (2006) seems to be wider. According to them, beacons are “key features in program code that serve as typical indicators of the program’s structure, operation or function”. That is, the presence of beacons provides supporting evidence for particular hypotheses in the source code.

Mapping from the high-level problem domain concept to the low-level solution domain characterizes Brooks' process as "top-down" (Cleary et al., 2008).

This process was further established by Soloway and Ehrlich (1984) when they found that expert programmers use top-down comprehension when they are familiar with small code fragments. They suggest that expert programmers use the following artifacts to decompose and map system goals into lower-level source code implementations, during both development and then maintenance:

- *Beacons* as described above

- *Programming plans*

Programming plans are generic program structures that portray stereotypical action sequences. These might now be referred to as "implementation patterns", analogous to "design patterns" (Gamma et al., 1994, Fowler, 2003), where the programmer uses stereotypical structures to achieve a certain functional or design goal.

- *Rules of programming discourse*

A programming discourse rule is a standard practice in programming. In other words, the communication conventions used by programmers implicitly when coding, such as standardized variable naming.

In addition, Soloway and Ehrlich (1984) referred to delocalized plans, which are programming plans that are scattered all over the programming codes. They proposed that delocalized plans would complicate the comprehension process.

Detienne (2002) tied the ideas of programming plans and hypotheses to the cognitive concept of schemas. According to Soloway et al. (1988), a schema is a knowledge structure or framework that represents stereotypical behaviour patterns (e.g., going to the petrol station, doing shopping). This framework can be used to explain and combine individuals' perceptions into a logical model (e.g., one can guess what is about to happen just by seeing a cue and having this schema at hand). For example, seeing someone in a petrol station queuing in front of a petrol pump might prompt a viewer to get into the queue behind them.

In contrast to top-down models, bottom-up models start from the source code and move upward to the problem domain. In other words, programmers start this comprehension approach from the code. These code statements are then grouped into higher-level abstractions. The aggregation to these code statements will continue until higher-level understanding about the program is achieved (Shneiderman and Mayer, 1979). This aggregation process will involve the syntactic and semantic knowledge of programs, where syntactic knowledge is language dependent and concerns the statements and basic units in a program. On the other hand, semantic knowledge is language independent, referring more to meaning, and is built in progressive layers until a mental model is formed which describes the application domain (Storey, 2005).

Pennington (1987a) presents a refinement of this bottom-up model. She found that programmers place an ordering on their abstraction process (which she refers to as "chunking"), first creating a "control-flow abstraction that describes the operation sequence of program". This is called the "program model". This program model later becomes the basis of the "situation model" that captures more delocalized functional abstractions and data flow abstractions. These latter abstractions are also possibly delocalized (Storey, 2005).

2.3.2 Encompassing Theories

There are a number of software comprehension models that try to bring the individual comprehension theories together. The most established of these are reported by Detienne and Von Mayrhauser.

Detienne's Comprehension Model

According to Detienne (2002), there are two different viewpoints in comprehending software: text understanding and problem-solving.

i. Text Understanding

In this perspective, there are three models that inform the techniques used by programmers to comprehend text (source code):

- *Functional Model*

The technique in this model is largely top-down and corresponds to the application of knowledge schema. Specifically, programmers will refer to knowledge schemas as they read text (the source code or the artifact) (Détienne, 2002). They match the content they find in the code / artifact (program structures, variable names, plans) to these functional schemas. For example, one of the scenarios for an ATM system is “withdrawing money”. This scenario consists of a sequence of reading bank card information, user verification, the user requesting the amount of money to be withdrawn, checking the account balance, and so on. In the context of the functional model, as the programmers read the related program source code, they will match it with this “withdrawing money” sequence. This overlaps with the Brooks model described above.

- *Structural Model*

This model employs a hierarchical breakdown of the program where programmers develop a network of relations to comprehend the program (Détienne, 2002). There are two models in this structural approach that inform the construction of this network of relations: namely, the structural schema approach and the propositional network. The structural schema approach is mainly top-down in orientation. According to Detienne (2002), “superstructures” (structural schemas) that describe the “general structure of stories” are useful as a guide in the comprehension process. This structural schema is invoked when programmers read source code (top-down) but in this instance it refers to structures like the data storage section in COBOL, header files in C or objects and object hierarchies in object oriented code.

Unlike the structural schema approach, the propositional network is bottom-up in orientation (Détienne, 2002). O’Brien (2007) assumed the “proposition” in this context is “a unit of information that contains one or more arguments along with a relational term”. He suggests that the propositions are connected to each other through referential links and, in order to understand a program,

programmers will have to identify and understand these referential links that reside in source code. An example would be where the programmer works to recreate call graphs or data slices (Gallagher and Lyle, 1991).

- *Mental Model*

According to Storey (2006), the mental model is programmers' mental representation of the program to be comprehended. This model is based essentially on Pennington's (1987b) work. It also can be described as a combination of the structural approach and functional approach. In this context, O'Brien (2007) suggests programmers have to build a thorough representation of the situation and the structure, interleaving them.

ii. Problem-Solving

Detienne's (2002) second perspective for program comprehension is in line with earlier work by Gilmore and Green (1984) that suggests the goal of program comprehension is to solve a problem (this can be directly related to maintenance or evolution). Likewise, Koenemann and Robertson (1991a) suggest this problem-solving perspective is a better framework than seeing program comprehension as a text understanding process. In this context, Littman et al. (1986) illustrated empirically that programmers tend to read source code either in a systematic (holistic) or as-needed (minimal) approach, as discussed below :

- *Systematic Approach*

In a systematic approach, the programmer will inspect the whole program to gain static knowledge that represents "what's in the function(s)" and dynamic knowledge that represents "delocalised run-time relationships" (O'Brien, 2007). This is done before any effort to modify the source code.

- *As-Needed Approach*

An as-needed approach means that the programmer will only concentrate on the code that is related to the current task and does not focus on the delocalized (dynamic) relationships in much detail. This approach seems

to minimize programmers' comprehension efforts as it consumes less time (and effort). However, Yong (1996) and Littman both suggest that one limitation of this approach is the possibility that working on specific, relevant code fragments might make programmers miss some of the more delocalized subtle dependencies in the code and therefore might introduce new problems.

Interestingly, Littman et al.s (1986) experiment involved a debugging task on a small program. The systematic approach might work well in comprehending such a small program but it may not be practical for large and complex programs. Hence, it is likely that the as-needed approach might be more relevant for larger systems.

Von Mayrhauser's Comprehension Model

In a related encompassing theory of comprehension, based on Letovsky's (1986) work, Von Mayrhauser and Vans (1995) integrated both top-down and bottom-up program comprehension into a unified model.

Letovsky (1986) observed activities called inquiries when programmers studied code. These were essentially programmers' information seeking actions of asking questions, speculating answers, and verifying answers by referring to artifacts such as the source code and software documentation. Based on his observations, Letovsky described the human "understander" as an opportunist processor able to utilize both bottom-up and top-down cues. He empirically derived this finding in a study using a think-aloud protocol where he identified interleaved "how", "why" and "what" questions that correspond to the comprehension strategies used by programmers. How questions (for example, "*How does the code calculate the Tax Free Allowance?*") could represent top-down strategies (Shaft and Vessey, 1995), while why questions (such as, "*Why is X 7 at this line?*") and what questions (for example, "*What does this piece of code do?*") reflect the source code to domain (bottom-up) rationale employed by the programmer (Détienne, 2002, Pennington, 1987a).

Von Mayrhauser and Vans (1995) proposed a cognition model that utilizes an opportunistic assimilation process similar to that mentioned by Letovsky (1987), based on studies of industrial programmers at work. Von Mayrhauser and Van's model combines a knowledge base identical to Letovsky's, with Pennington's program and situation models and the top-down process suggested by Soloway and Ehrlich (1986). The combination produces a model comprised of four major components. As illustrated in Figure 2.1, comprehension is performed at various levels of abstraction concurrently and this is done by switching between the three comprehension processes included (top-down, situation and program models).

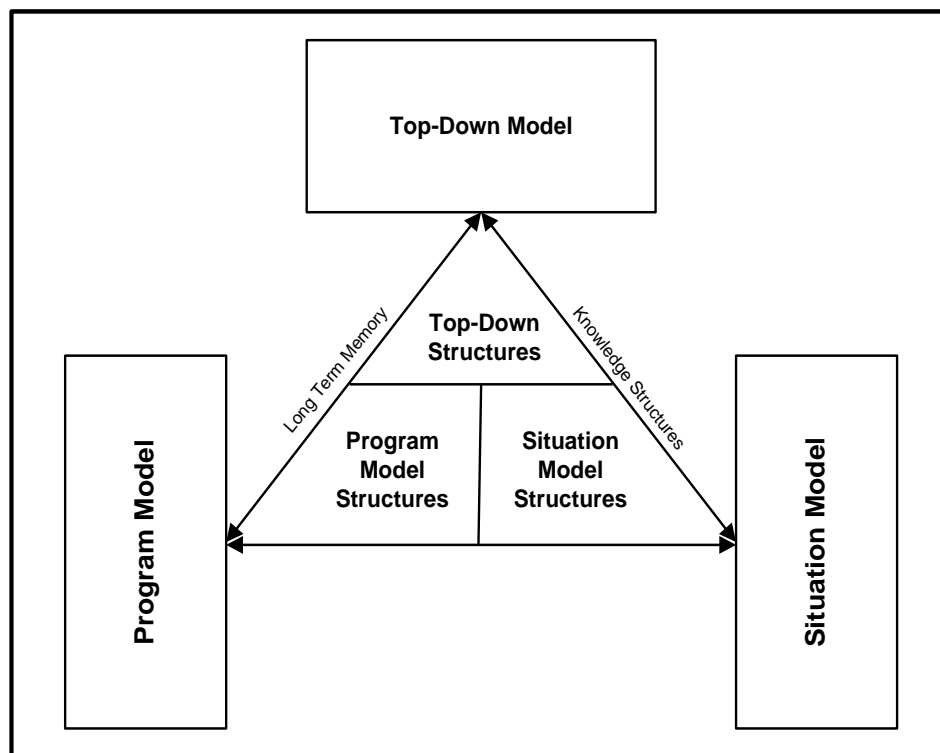


Figure 2.1 - Meta-model of Von Mayrhauser and Vans (1995)

In Figure 2.1 above, the three rectangular components represent the involved comprehension process in constructing mental representation at different abstraction levels. Meanwhile, the triangular component (at the centre of the diagram) represents the knowledge base developed and enhanced by the comprehension process. Von Mayrhauser and Vans (1995) explain that:

- The programmer will normally turn to top-down comprehension when they are familiar with the system. In this approach, the domain knowledge serves as a starting point to formulate hypotheses.
- In a situation where the programmer is unfamiliar with the application domain and the code, he/she will shift to the program model. The program model is concerned with generating control flow-based abstractions and text representation-based abstractions.
- The situation model (that increasingly drives data flow and functional abstractions of the program) is augmented after a (partial) program model is constructed.
- Programmers' current knowledge is represented by the knowledge base. As the comprehension process progresses, this knowledge base will store the deduced knowledge and also supply required information towards the construction of the three cognitive models.

In this model, Von Mayrhauser and Vans (1995) suggest that programmers utilize various comprehension strategies to reach a specific goal. They will invoke a particular comprehension model based on their current context including familiarity with the specific code they are looking at, the information stored in the knowledge base and the problem that they are addressing during the comprehension process. Von Mayrhauser and Vans proposed that “unlike the component models taken in isolation, the integrated model allows activation of and switches between any of the three component models at any time during the comprehension process”.

2.4 Information Seeking and Software Comprehension

As the previous review illustrates, in early works, researchers concentrated on the cognitive aspect of software comprehension. These works was characterized by the development of cognitive models which were difficult to embody effectively in software tools or to characterize as guidelines for effective software

maintenance (Buckley, 1994). More recent work has focused on a more-applied information seeking perspective and this section briefly reviews works that reflect this evolution.

In the context of software maintenance, information seeking is known as a fundamental sub-task in understanding programs (Curtis et al., 1988; Seaman 2002; Singer, 1998; Sim 1998; O'Brien et al., 2006). While information seeking seems to closely connect with programmers' tasks, only a small number of studies have been undertaken to explore how programmers actually gather this information. However, an increasing number of researchers, such as Seaman (2002), Ko et al. (2007) and O'Brien (2007) are now addressing this area and information seeking is now seen as a credible perspective when investigating software maintenance behaviour.

2.4.1 What is Information Seeking?

Information seeking has been defined as the searching, recognition, retrieval and application of meaningful content (Kingrey, 2002). According to Kingrey (2002), information seeking has been accepted as a basic human behaviour in cognitive exercises, in social and cultural activities (the exchange of ideas) and as a strategy to handle in-doubt situations.

Information seeking is typically viewed by researchers as a problem-oriented action, thus explicitly tying this line of research to Detienne's problem-solving perspective for program comprehension. For example, O'Brien (2007), Case (2007), Dewey (1933) and Marchionini (1997) agree that the main purpose of information seeking is to solve problems. Indeed, Marchionini (1997) defines information seeking as a "problem oriented process" that is "closely related to learning and problems".

In recent years, an "information behaviour" label has also been used as an umbrella term for a broad range of information-related phenomena (Case, 2007). In fact, according to Niedźwiedzka (2003) information behaviour can be seen as

a more encompassing term than information seeking in the context of problem-solving, as shown in Figure 2.2 below.

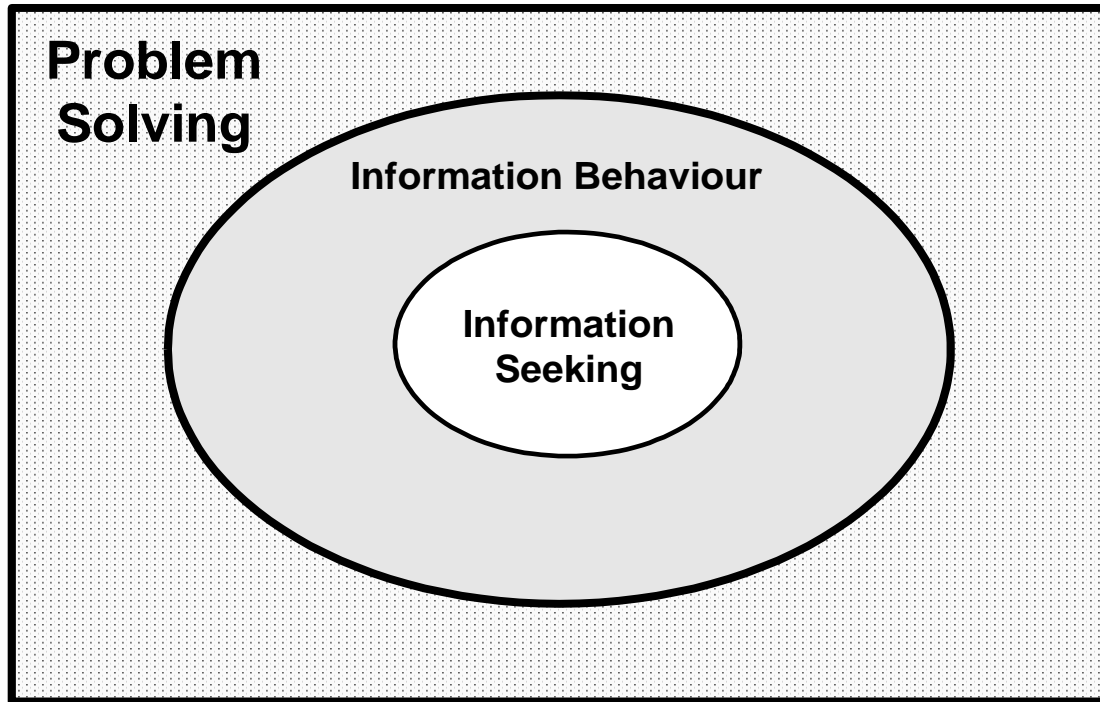


Figure 2.2 - Information seeking in context (Niedźwiedzka, 2003)

In this research, the problem-solving task is a maintenance one. Information seeking is an activity towards understanding the systems, and seeing all possible options available to the maintainer.

2.4.2 Information Seeking Models

Krikelas (1983) suggests that the information seeker will start the information seeking process as they become aware that their current state of knowledge is insufficient to deal with a certain issue. The resultant information seeking process may result in the fulfilment or non-fulfilment of this insufficiency. The seeking process might be repeated until the information seekers' need is fulfilled as demonstrated by O'Brien (2007) and Hayden (2001); that is, failure to find the

appropriate information may lead the information seeker to repeat the information seeking process.

Significant interest has been shown in informing conceptual models or frameworks of information seeking and retrieval (Jarvelin and Wilson, 2003). This section discusses several of these early information seeking models such as the models proposed by Ellis et al.(1997), Kuhlthau (Kuhlthau, 1988, Kuhlthau, 1993), Marchionini (1997) and Wilson (Wilson, 1981, Wilson, 1997, Wilson, 1999). Due to the lack of work in information seeking in software maintenance, most of these early models are from other domains and not designed for a software maintenance setting. However, in more recent work (O'Brien and Jim Buckley, 2005, Buckley et al., 2006, O'Brien, 2007), these models were adapted and integrated into a behavioural model of the information search and retrieval processes of maintenance programmers.

To summarize these early models from other domains, Figure 2.3 shows the models placed side by side. Here the horizontal dimension can be read as “similarity of stage”. These models are placed side by side to emphasize the similarities between them. For example, Ellis et al. refer to sourcing information and chaining of these sources (through mechanisms such as referencing in articles). Analogously, Marchionini discusses choosing a search system and a query to find information sources, while Wilson and Kuhlthau refer generically to “problem resolution” and “information collection”, respectively.

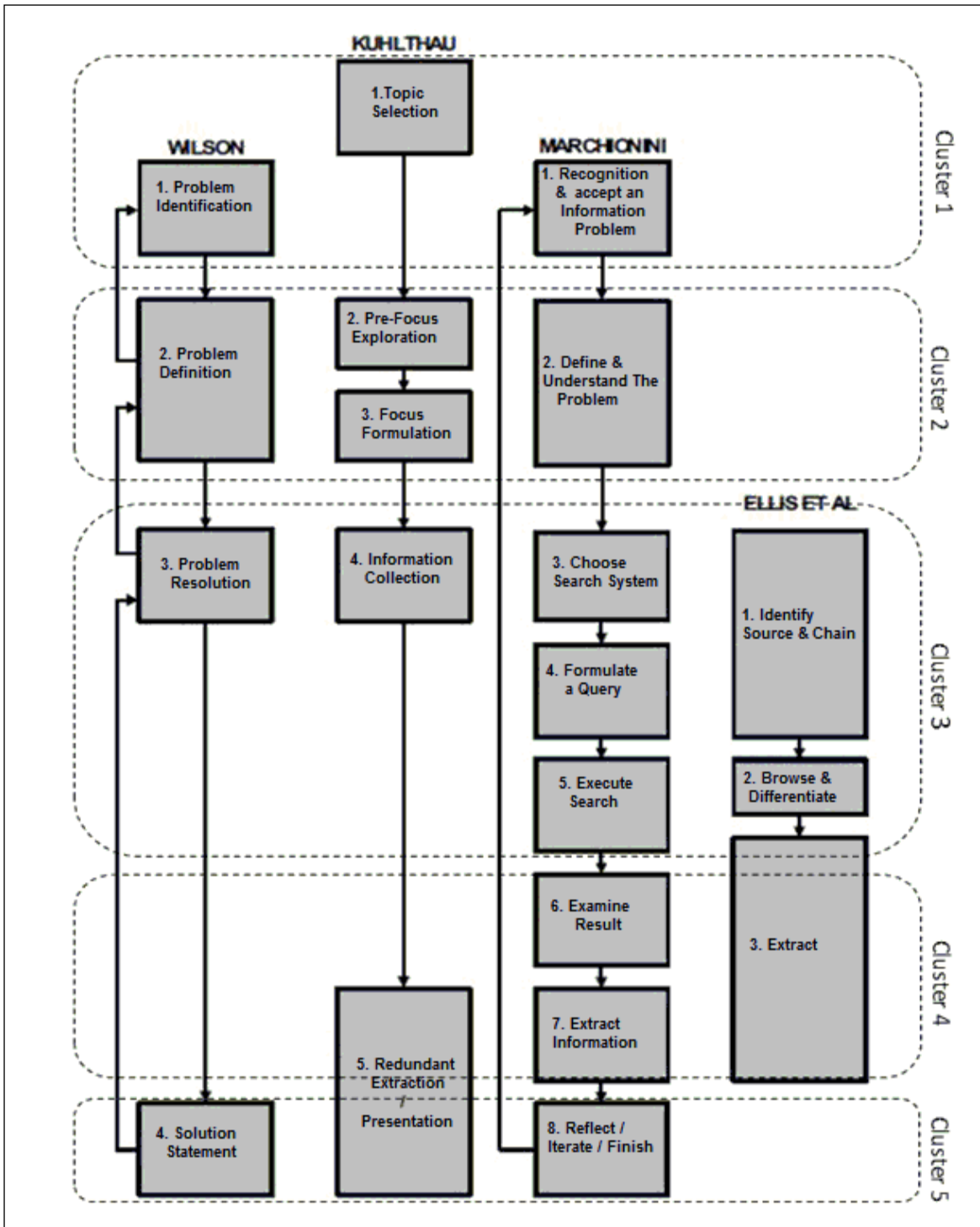


Figure 2.3 - Correlation of information seeking theories, adapted from O'Brien (2007)

The clusters in Figure 2.3 represent the stages of information seeking as follows:

- Cluster 1: *Awareness*. This is the stage where the information seeker becomes aware of the problem (information needs). Wilson (1999) called this “problem identification” where the information seeker becomes aware of the problem and starts to identify the problem by asking questions such as “What sort of problem is this?”
- Cluster 2: *Defining the Problem*. The information seeker will then determine the sought-after information. Based on the problem that was identified in the previous stage, Marchionini (1997) suggests that the information seeker will attempt to understand the information need along with defining the problem. Marchionini notes that this activity will remain active as long as the information seeker is seeking information.
- Cluster 3: *Searching*. Kuhlthau (1988) refers to this stage as “information collection” where information on the defined focused topic is gathered. Ellis et al. proposed a two-stage process for this cluster in the context of research: identify source and chain, and browse and differentiate.
- Cluster 4: *Result Assessment*. At this stage, according to Marchionini (1997), the information seeker examines the obtained result and evaluates them in terms of adequacy, relevance, type and format of the findings related to the problem in hand.
- Cluster 5: *Prompted Action*. Wilson (1999) and Kuhlthau (1988) suggest that the information seeker is now satisfied and uses the findings. However, Marchionini (1997) and Wilson (1999) discuss the possibilities for the information seeker to reflect on their findings and decide to finish or repeat the seeking process.

While these models provide a general information seeking process, all of them were designed for other domains and were not specifically designed for software maintenance. O'Brien (2007) addressed this by aggregating these models into one and evaluating the resultant model in a software maintenance context empirically.

2.4.3 Information Seeking in Software Maintenance

O'Brien and Buckley (2005) and O'Brien (2007) proposed a five-stage, non-linear, iterative information seeking model (ISM) for software maintenance based on the information seeking models discussed previously and subsequent empirical evaluation using software practitioners in-vivo.

In this model, each stage is freely invoked (non-linear) at any time and the programmer may execute the process at any point and shift to different stages according to their current knowledge. The authors did suggest that stages would be largely invoked linearly from top to bottom and the empirical data did show this for a number of transitions. The proposed ISM is shown in Figure 2.4.

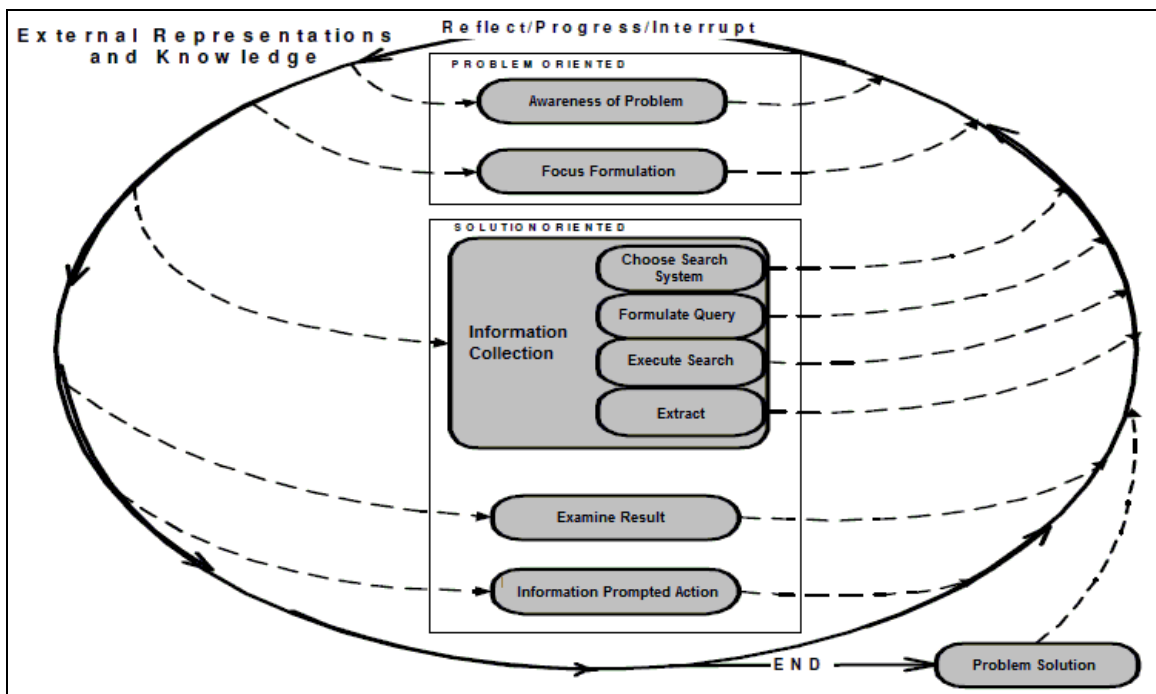


Figure 2.4 - ISM for software maintenance (O'Brien, 2007)

Reflecting previous research in software comprehension, O'Brien (2007) divided his model into two main phases: the problem-oriented phase (top-down comprehension), and the solution-oriented (an approach with possible bottom-up and top-down comprehension elements). The non-linear, iterative flow, based on a mixture of reflection, progression and interruption, represents the combination of these two comprehension strategies and their sub-elements.

The first phase is the problem-oriented phase. This is the starting stage when the maintainer or programmer becomes aware of the problem and decides to refine their understanding of the problem. This stage is divided into two sub-stages:

1. Awareness of problem: This stage is invoked when the programmer starts to become aware of the problem and constructs an early understanding of the problem. The programmer will identify the problem and ask questions such as "What sort of problem is this?".
2. Focus formulation: The programmer will try to get a tighter understanding of the problem by formulating specific queries (such as "Why /How/ Where did this problem happen?") as an agenda for the next stages.

The solution-oriented phase starts when the programmer moves to find information towards a solution. It consists of four stages:

1. Information collection: During this stage, the programmer attempts to address the formulated queries from the last two stages (in the problem-oriented phase) by identifying, browsing and extracting information from various sources (and in various forms). It consists of:
 - a. Choosing a search system such as lexical matches in the source code IDE, or documentation. It is guided by the programmer's knowledge about the application domain and system.

- b. Formulating a query where the programmer matches his/her understanding of the problem with the selected search system.
 - c. Executing a search where the programmer searches for information within the search system.
 - d. Extracting information where the programmer takes the information found in the search system by reading or scanning it. They may also classify, copy and/or store the relevant information at this stage.
2. Examination of results: At this stage, the programmer will inspect the results from the previous stage. These are examined in terms of quantity, type, relevance and format.
 3. Information-prompted action: At this stage, the programmer will feed back into the process cycle for further information seeking activities, or will proceed to problem solution.
 4. Problem solution: This is where the programmer is satisfied that the problem has been solved.

2.5 Information Seeking-Related Issues for Maintenance Programmers

The research in this thesis focuses on the information collection stage of O'Brien's (2007) information seeking model where information is gathered by programmers in an OS team, as part of their maintenance activities. As we can see from the models on which O'Brien's (2007) ISM is based, typically, the information seeking process is seen as an individual activity. However, in many situations, people need to collaborate with each other as they are working in a group and sharing the same goal / task (e.g., programmers in a maintenance team). Hence, information seeking can also be considered as a collaborative activity (Ko et al., 2007). This work extends O'Brien's work in this fashion.

Similar to individual information seeking, the team must be aware of its information needs. To do this in a collaborative environment, the team members must effectively spread (communicate) the retrieved information to make sure that all of the team members have the same awareness (Ko et al., 2007). This is in line with the approach of Poltrock et al. (2003) who extended the definition of information seeking to include a communication element. According to them, information seeking requires communication about information need, information sharing among the team members, and coordination of information retrieval activities across team members. This work explicitly addresses this issue.

This study looks at open source programmers who choose the developer mailing lists as their search system. They formulate (and explicitly state) queries in their emails to this group. Then they execute their search by sending the email to the group. They then extract information from the responses. Thus, the information medium is the mailing list.

This work will focus on the artifacts they seek information about, and the type of information they seek from these artifacts, as detailed in their emails. Subsequently, the quantity of response they obtain from their community is evaluated. Given these agendas, we now look at research that addresses programmers' information source: that is, the artifacts programmers look for information about. Note that this is not the media through which they get this information (in our case, the mailing list). Rather, we look at research addressing the information that programmers seek from distinct artifacts.

2.5.1 Information Sources

In studying the work practices of software maintenance engineers, Singer (1998) interviewed two engineers who worked on the same system. In expanding her samples, the same interview was done at 10 different companies. Her study suggests that:

- Programmers regard source code as a primary source when they are doing enhancements to programs. This finding was also reflected in later studies by Seaman (2002) and De Souza (2005).

Interestingly, Littman et al. (1986) and Koenemann et al. (1991b) refined the idea of the importance of code. They found that as “program comprehension is best understood as a goal-oriented, hypotheses-driven problem-solving process”, programmers will employ an as-needed strategy when viewing code (see section 2.3.2). Hence, they restrict their study of the code to those parts that they find relevant for a given task, and ignore other parts of it (Koenemann and Robertson, 1991b). A good example of this is Mylar (Kersten and Murphy, 2005), an Eclipse IDE plug-in, that emphasizes programmers’ interest in particular parts of the code base (based on their past viewing patterns).

- Programmers only occasionally refer to documentation, as they believed that documentation is possibly inconsistent. Their distrust in documentation is possibly because programmers regard the activities of creating and maintaining software documentation as time consuming. Hence, they assume that the documentation they might consult is improperly done or incomplete.

Seaman (2002) also suggested other information sources that programmers refer to while evolving their systems. Seaman noted that while programmers put the highest preference on source code, they also valued execution traces and trusted colleagues. Execution traces are claimed by programmers (who regularly used it) to be the most efficient and accurate way to understand a problem, particularly when used concurrently with program source code.

Seaman (2002) also found that programmers tend to consult with other in-house programmers working on the same system. Seaman described this human information source as an “accurate” and “crucial” source of information. This finding supports our protocol, where a mailing list discussion between

programmers was observed as the media through which OS programmers resolved issues that they could not resolve with their current information sources.

Seaman's (2002) study was, however, done in a commercial software development setting where programmers, documents and all the required tools were normally co-located in the same place. It should be acknowledged that these findings could be different if this study was done on globally distributed OS software maintainers.

Interestingly, in Seaman's (2002) study, programmers reported that sometimes information sources were not available or accessible. For example, the original programmer of the code may have left the organization. This is an example of "information blocking" – a situation where programmers fail to access the required information source when they require it. Bradac et al. (1994) did some related research in the area of information blocking. Their study found that information blocking can be very time-consuming, taking up to 60% of the allocated time for software maintenance.

2.5.2 Information Sought

Jarvelin and Repo (Jarvelin and Repo, 1983, Jarvelin and Repo, 1984) suggest that the information sought by people can be categorized into three categories, namely, information about the problem, the domain of the problem, and problem-solving. These categories were later proposed by O'Brien (2007) as pertinent to the software maintenance task, and therefore, applicable to programmers:

- Problem information

This category of information represents the structure, properties, and requirements of the current problem.

- Domain information

This category of information describes the surrounding area of the problem or information that is related to the area of the problem. It

consists of known facts, concepts, laws and theories on the domain of the problem.

- Problem-solving information

This category of information describes generalized ways of solving the problem such as:

- how problems should be seen and formulated
- what problem and domain information should be used
- how it should be used in order to solve the current problem.

There have also been several more directly relevant empirical studies that characterize the types of information sought by programmers in the context of software comprehension (Letovsky, 1986, Pennington, 1987a, Corritore and Wiedenback, 1991, Good, 1999, O'Shea, 2006, Sillito et al., 2006, Ko et al., 2007, Sillito et al., 2008) These studies focus on the information that programmers need and the information that they find difficult to obtain during software maintenance, thus potentially informing the design of software visualization tools.

Several of these studies focused their efforts on small programs or on student programmers (such as Letovsky, (1986) Pennington, (1987a); Good, (1999). For example, in landmark studies by Pennington (1987a, 1987b) five information types available from the source code were identified:

- *Function* is the information about the overall goal of the program. For example, “What is the purpose of the program?”, “What does the program do?” Functional information also includes the sub-goals of the program. Some information about which events preceded others can be deduced from this perspective but not the detailed implementation of the events.

- *Control flow* is the information about the temporal sequence of events occurring in the program. For example “What happened after X occurs?”, “What has occurred just before X?”.
- *Data flow* is essentially concerned with the transformations that happen to data objects during execution. This includes data dependencies. For example, “Does variable X contribute to the final value of Y?”
- *Operations* refers to information about a single line of code or less, such as “Does a variable become instantiated with a particular value?”.
- *State* refers to the connection between execution of an action and the state of the program that is necessarily true at that point. For example, “When X is equal to 65, does event Y take place?”.

Most of the studies reporting on detailed information types sought are derived from this original “information-types” schema developed by Pennington (Pennington, 1987a, Good, 1999, O’Shea, 2006). However, as this schema was developed through a theoretical review of the information available in segments of code, it is possible that it ignores other artifacts produced by the development team and that it ignores some information seeking requirements specific to larger code bases (Sharif and Buckley, 2009a). An illustrative example is the ‘location’ information type identified by O’Shea (2007), where programmers sought the location of a specific piece of code within the software system, again reflecting Littman et al.’s observation of the “selective comprehension” behaviours of programmers when faced with code irrelevant to their task context. The need for location information would not have been apparent in small code segments.

An alternative approach is that of Ko et al. (2007) who reported on commercial software development in co-located teams. They carried out a two-month field study (25 hours observational work) of 17 programming groups at Microsoft to identify the information software developers seek, where they find this information, and what may inhibit the acquisition of such information.

Ko et al. (2007) used an open coding protocol to characterize programmers' information needs as they maintained commercial software. They identified 21 such information needs (in the context of seven maintenance tasks). The most prevalent were:

- Information on the ripple effects of their changes;
- Information on the causes of specific program states and bugs and;
- Information on the changes performed by colleagues.

They also went on to identify the information types that programmers had difficulty obtaining during maintenance and, by means of a survey, the information types that the programmers thought were important. They found that programmers thought that it was important to know:

- the causes of a specific program state or failure
- the program's goals
- the implications of a change
- what a specified code failure looks like.

Programmers found it particularly difficult to obtain information on the causes of specific program states and the ripple effects of changes they had made. Given that both these types of information were frequently sought and thought of as important, these particular difficulties should be of interest to tool-builders.

Similar with Ko et al (2007), Sillito et al (2008, 2006) undertook an open coding protocol to study programmers' information seeking while they are performing change task. Specifically, they wanted to characterize code-based information sought by programmers when evolving a software system, how they used tools in seeking the information and how well the existing tools supported the information seeking activities. They reported on code-change tasks done by co-located programmers in two different setting: the laboratory and in an industrial setting. They identified 44 types of questions asked by programmers while they were making changes and organized them into four levels of question categories. In

contrast with Ko et al (2007) who focused on wider range of activities and broader context, Sillito et al 's (2008, 2006) study focused on more detailed activities: namely the code changing task. Hence, it resulted in a different range of information needs, the most prevalent of which were:

- Information on locating points in the code relevant to the task. This served as a starting point for programmers to work with the code and continuously explore the codes by finding another focus point.
- Information that relate methods, types, objects and entities to domain concepts or system behaviour.

The information needs mentioned above were placed in 4 different stages. Below are the 4 stages of question asked by programmers (Sillito et al., 2008, Sillito et al., 2006) :

- i. *Finding Focus Points.* Questions that focus on relevant point to start the change task. This could include location, methods, types or any entity in codes. For example, "Where in the code is the text in this error message?"
- ii. *Expanding Focus Points.* Questions to expand the information derived from the *Finding Focus Points* task. Often, this is done by exploring relationships between the entities. For example, "What data is being modified in this code?"
- iii. *Understanding the structure.* Questions that help build an understanding of concepts in the code. This involves the multiple relationship and entities resultant from *Expanding Focus Points*. For example, "How are these types or objects related?"
- iv. *Questions over Groups of structures.* This stage involves understanding the relationships between multiple structures. For example, "How does the system behaviour vary over these types?"

It should be noted that, given the potential “blinkering” effect of adhering to a pre-established schema such as Pennington’s information-type schema, it is proposed that the study of information seeking behaviour in this thesis should be inductive, in the manner of Ko et al. (2007) and Sillito et al (Sillito et al., 2008, Sillito et al., 2006), where the data is analysed and the categories of information sought should be derived from the data without a pre-established framework.

2.5.3 Concept / Feature Location

Requests for (software evolution) changes are normally formulated in terms of domain concepts (Rajlich and Wilde, 2002). For example, a change request could be to “add credit card payment to the point-of-sale system”. The maintainer is then required to understand where and how the related concepts are implemented in the source code. This phenomenon is called concept/feature location. In this context, Rajlich and Wilde (2002) described the concept/feature location as “the starting point for the desired program changes”.

Concept location can be defined as “a process that maps domain concepts to the software components” (Chen and Rajlich, 2000). This process is illustrated in Figure 2.5. The diagram shows that the input for concept location is typically a change request that is described in “natural language” and expressed in “domain level terminology”. During the concept location process, the change requests (concepts) are then mapped to the related components (in the program’s source code) that implement the particular concept. Therefore, the output of this process is a set of program components that implement the concept or feature (Rajlich and Wilde, 2002): essentially an output defining the relevant code. Hence, concept location can also be considered as a part of the incremental change process (Rajlich, 2004) and thus a maintenance-relevant information seeking exercise.

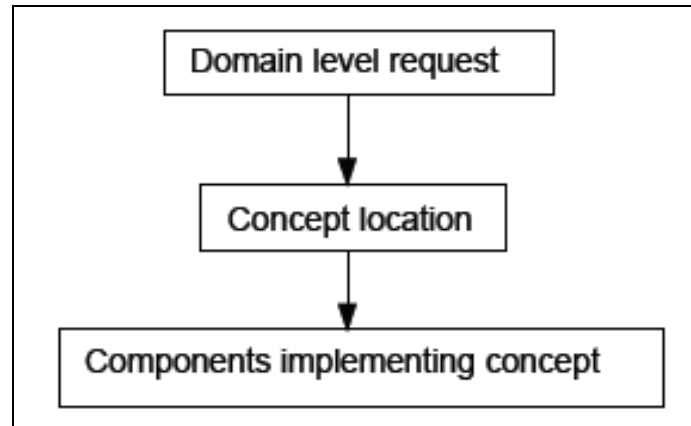


Figure 2.5 - Concept location process (Chen and Rajlich, 2000)

Concept location can be classified into two sub-types:

- Feature location, which refers to seeking observable user functionality (Wilde et al., 1992). According to Poshyvanyk et al. (2007), a feature in a program represents some functionality that is visible and accessible by programmers. During software evolution, it is important for programmers to identify the specific locations in the source code that are related to particular functionality. This is considered as one of the most common tasks of programmers (Poshyvanyk et al., 2007).
- Concept location is a superset of Feature location, concerned with identifying the functional requirements, non-functional requirements and other structuring of the system. Concept location then refers to identifying a mapping between all or any of these concerns and the code body. For example, a programmer trying to separate the system from the GUI might try to remove the GUI from the rest of the system and instead provide an API for other systems. Here the concept is GUI – not a user functionality but still an evolutionary concern (Buckley, 2010).

Concept location plays an important role in understanding programs. Lakhoria (1994) claimed that, in many cases, it is unnecessary (and often impossible) for programmers to comprehend the whole program source code. Instead, programmers – especially experienced programmers – can employ the as-needed strategy where they only have to comprehend parts of the source code

that reflect specific concepts (Koenemann and Robertson, 1991b). Thus, they are minimizing the effort to comprehend the program by seeking understanding that is just enough (minimum) for their current task (Lakhotia, 1994). Hence, concept location theory suggests that programmers seek relevant information during software maintenance, a finding that reflects Littman et al.'s (1986) initial observations. This, in turn, reflects the information seeking activities during software maintenance.

Feature or concept location is relatively easy in small programs, where the programmer possibly understands the whole program. However, it can be a difficult task for large and complex systems (Chen and Rajlich, 2000). Consequently, several researchers have tried to use information retrieval (IR) techniques to support this activity (Poshyvanyk and Marcus, 2007, Cleary et al., 2008). For example, (Cleary and Exton, 2007) and (Cleary et al., 2008) used a combined IR-based approach to handle the concept location problem. Unfortunately, automated concept location of this sort can still only be realized to low levels of accuracy (Cleary et al., 2008) where accuracy is measured by precision and recall.

2.6 Information Seeking in Open Source Software Development

While the works discussed in the previous section identify information seeking as a core software maintenance concern, our literature review suggests that, to date, there is little research to inform on information seeking among OS programmers (Sharif and Buckley, 2008a, Sharif and Buckley, 2008b, Sharif and Buckley, 2009a, Sharif and Buckley, 2009b). It is important to study software maintenance in an OS context because, as discussed above, OS software is prevalent, trusted and relied-upon. Like commercial software development, OS developers deal with large scale code with high code complexity (Daniel et al., 2009b).

This section expands on the context within which information seeking is studied in this thesis: namely, OS software evolution. It identifies the core characteristics

of OS development, illustrating differences that suggest different information seeking processes may be at play in that context. Finally, it will look at where OS programmers may source the information they need, thus guiding our empirical work.

2.6.1 Open Source Definition

The Open Source Definition (OSD) document created by the Open Source Initiative (2009) provides the basic requirements for projects to qualify as open source (Figure 2.6). Thus, the distribution terms of open source software must comply with criteria presented in that Figure.

There is a general assumption by the community that the criterion for OS projects is the availability of their source code. However, according to the OSD, the open source concept is not limited to access to the source code. In fact, OS has three core criteria (Feller and Fitzgerald, 2002):

1. The OS software can be distributed freely
2. The source code can be distributed and modified
3. Work derived through software modification can be created and distributed.

The other seven criteria emphasize licensing issues and the “no discrimination” policy (Gacek and Arief, 2004) which allow anybody to use the software in any field or area.

1. Free Redistribution

The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.

2. Source Code

The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost, preferably downloading via the internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

3. Derived Works

The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

4. Integrity of the Author's Source Code

The license may restrict source code from being distributed in modified form *only* if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

5. No Discrimination against Persons or Groups

The license must not discriminate against any person or group of persons.

6. No Discrimination against Fields of Endeavour

The license must not restrict anyone from making use of the program in a specific field of endeavour. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. Distribution of License

The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. License Must Not Be Specific to a Product

The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

9. License Must Not Restrict Other Software

The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open source software.

10. License Must Be Technology-Neutral

No provision of the license may be predicated on any individual technology or style of interface.

Figure 2.6: The open source definition by Open Source Initiative (2009)

2.6.2 Open Source Software Development Process

According to Feller et al. (2002), there are seven common characteristics of an OS software development process. They propose that the generic OS development process:

- 1) is parallel, rather than linear;
- 2) involves large communities of globally distributed developers;
- 3) includes the participation of highly talented, highly motivated developers;
- 4) includes increased levels of user involvement;
- 5) utilizes truly independent peer review;
- 6) provides prompt feedback on user and developer contribution;
- 7) makes use of an extremely rapid release schedule.

Parallel Development

Parallel development refers to the practice of OS programmers working concurrently on the same system. This especially occurs when OS programmers are working on a large system. In this approach, OS programmers (individuals or in groups) are working in parallel on different aspects of the large system (Feller and Fitzgerald, 2002) and this allows OS programmers to work simultaneously rather than waiting on each other. Hence, it makes it possible for large OS programmer community members to collaborate efficiently.

In this context, good task coordination between OS programmers is vitally important to success (Bonaccorsi and Rossi, 2003) and this suggests that maintaining awareness of other programmers' activities is even more necessary than in co-located development (Ko et al., 2007). Indeed, Gutwin et al. (2004), stress that awareness is important for distributed programmers and that they have to maintain a "general awareness" that every team member should share. However, the same authors state that information on how OS teams manage their collaboration seems to be very limited.

Large, Globally Distributed Development Communities

The OS programmer communities involved in OS software projects are potentially globally distributed (Feller and Fitzgerald, 2002). For example, Dempsey et al. (2002), in their study of MetaLab Linux Archives, found that 2429 different programmers contributed to the project. The programmers came from at least 71 different countries (including the US, Germany, the UK, Holland, Australia, France, Italy, Canada, Sweden, Finland, Austria and the Czech Republic). Likewise, Hertel (2003) reported that 1200 programmers from at least 28 countries were involved in the Linux kernel (2.2.10) project.

These globally distributed programmer communities need a practical and efficient medium to interact and collaborate. OS programmers utilize a wide variety of internet technologies for this purpose (Wu et al., 2007), primarily through text-based communication such as email and chat systems (Gutwin et al., 2004). Consequently, there are possible implications for OS programmers' information seeking, the most obvious being the physical distance between programmers who might provide information. In other words, if programmers do decide that they need to look outside of the code for their information needs, as has been reported (Singer, 1998), it would seem more difficult to do so.

Highly Talented, Highly Motivated Developers

OS programmers are associated with high levels of ability and commitment (Feller and Fitzgerald, 2002). Fitzgerald (2004) describes OS programmers as "code gods acknowledged to be among the world's most talented and highly motivated programmers". While it is unlikely that all OS programmers are of this calibre, many of them are computer hobbyists who would gladly spend their spare time programming (Bonaccorsi and Rossi, 2003). OS allows these individual programmers to pursue their hobby in a meaningful way, that is, by participating in a large programming community (Dempsey et al., 2002). This

could be related to the hacker culture, as mentioned by Raymond (2001a). Hackers look on programming as an art form that gives them “artistic satisfaction associated with solving complex computer problems” (Bonaccorsi and Rossi, 2003).

Interestingly, the prevalence of these expert, motivated programmers could be a limiting factor in this research; that is, results from this work might not be translatable to a commercial setting where programmers may be less motivated. There is also the possibility of a lesser proportion of “code gods” in commercial software development, although this claim would need to be empirically validated.

Alternatively, these code gods who actively contribute to OS software development may also be working in commercial software projects. In this context, Bonaccorsi and Rossi (2003) suggest that one of the motivations for programmers to contribute to OS software projects is to gain a better chance of being noticed by other software firms.

However, according to Feller and Fitzgerald (2002), there are also members of the OS team who are novice programmers with less experience. Their participation does not necessarily slow down the overall progress of the project. According to Mockus (2000), these less experienced programmers normally tend to contribute in areas appropriate to their skill level. Hence, they are not impacting negatively on the overall progress of the project.

Actively Involved Users

Users seem to play an important role in OS software development, far more than is reported in proprietary software projects (Bonaccorsi and Rossi, 2003, Feller and Fitzgerald, 2002). According to Schmidt and Porter (2001), OS users can be involved in important tasks which “programmers are traditionally reluctant to fulfil” such as, for example, elaborating requirements, testing the functionality and writing documentation. This valued customer role is also evidenced by

arguments in (Raymond, 2001a) and by OS project websites where there are explicit sections for the users to contribute. Examples like Ubuntu³ and Jakarta⁴ project. illustrate this point. In projects where active users fulfil such roles, it is likely that certain documentation is more up to date than is typical in commercial software development and that developers may learn to trust this documentation more, especially seeing as they are largely confined to asynchronous communication.

Investigation of these websites also suggests that reference or documentation is important in allowing the users' participation in OS software development. This is in line with Daniel et al. (2009a) who suggest that one of the characteristics of successful OS projects is providing complete information targeted toward users and general information about the project.

Truly Independent Peer Review

It is expected that peer review among large and distributed development communities is truly independent. In a proprietary software development setting where all programmers belong to the same organization, it might be difficult to give a genuine criticism of other programmers' work, due to the "political and competitive nature of the workplace" (Feller and Fitzgerald, 2002): "even with the best will in the world, (same-organization programmers) may not be motivated to dig deep and find errors".

In the OS development setting however, programmers have no reason to artificially confound the peer review process since (in many cases) they have never met each other (Feller and Fitzgerald, 2002). Thus, feedback is likely to be genuine.

³ <https://wiki.ubuntu.com/ContributeToUbuntu>

⁴ <http://jakarta.apache.org/>

Prompt Feedback

Feller and Fitzgerald (2002) regard prompt feedback as one of the important factors in the success of peer review in OS software communities. In this context, Jorgensen (2001) describes the feedback activities in OS development as a constant cycle of “see bug, fix bug, see bug fixed in new release”. Schmidt and Porter (2001) also reported on the promptness of feedback in OS communities. They reported that:

Often it's only matter of minutes or hours from the time a bug is reported from the periphery to the point at which an official patch is supplied from the core to fix it.

It would be interesting to assess if the immediacy of response holds true for OS maintenance in this study. If so, it would suggest quick responses to questions and requests for reviews of alterations. However, maintenance is typically seen as a less attractive activity by programmers and in OS, where information may be sought by non-personal communication, the prompt feedback of information may suffer.

Rapid Release Schedule

All the characteristics discussed above — huge development team, promptness of feedback, parallel development and highly motivated programmers – support a rapid incremental release pattern (Niels, 2001, Raymond, 2001a). In this context, Feller and Fitzgerald (Feller and Fitzgerald, 2002) report that, in 1991, new versions of the Linux kernel were being released at the rate of more than one per day.

2.6.3 OS Programmers' Communication Channels

As mentioned above, OS programmers interact and collaborate using a wide variety of internet technologies, primarily through text-based communication channels such as discussion forums, chat systems and email (Angioni et al.,

2005). Chat systems would require OS programmers to communicate in real-time and may give the added benefit of getting a response fairly quickly. However, OS programmers are often geographically distributed all around the world and hence they work in different time zones. This would limit the number of participants for discussion at a particular time (Fernanda et al., 1999). Also, intuitively, chat is a more personalized form of communication and may be less prevalent than more formal communication mechanisms in distributed OS projects where programmers are less likely to know each other as well.

Discussion forums and mailing lists seem more suitable for OS project discussions. Mockus et al. (2002) describe discussion forums and mailing lists as part of the OS practice whereby programmers “work in arbitrary locations, rarely or never meet face to face, and coordinate their activity almost exclusively by means of email and bulletin boards”.

Discussion forums or message boards are online discussion locales where conversations are held in the form of posted messages. They are different to chat systems as they can be archived for long periods and need not be synchronous. Hence, it is possible to reach a larger audience to participate in discussions. However, message boards are normally presented as an open discussion and are organized according to topic (thread) with less emphasis on timeline or focusing on particular recipients. This makes the message board less suitable in situations that need coordinated work or to keep track of specific other co-workers or team members (Gutwin et al., 2004).

On the other hand, mailing lists have similar features to message boards in that they keep conversations in archives and are also accessible to a large audience. Mailing lists seem to be more suitable for work coordination and to keep track of others. Specifically, mailing lists can be targeted at specific persons and, at the same time, are still available to be read by others. Consequently, Gutwin et al. (2004) found that mailing lists are successfully used to gather and provide awareness among OS programmers. In fact, in their study, Gutwin et al. found that OS programmers could gain considerable implicit information just by

“overhearing” the email conversation between other individual programmers. This suggests that mailing lists are effective in maintaining awareness and serving as an announcement mechanism to the community. More holistically, Mockus et al. (2002) and Gutwin et al. (2004) suggest that mailing lists are one of the primary communication channels for OS programmers. The mailing list can be seen as a rich source of data as it contains a “substantial proportion of the information” that passes among globally distributed OS programmers (O’Shea, 2006).

2.6.4 Categorizing Open Source Projects

In an attempt to maximize the coverage of this work, it is important to address the different types of OS projects. Hence, it is helpful to identify an encompassing taxonomy of OS systems. There are only a few studies which categorize OS projects from different perspectives, one being Feller and Fitzgerald’s (2002) original study that has been referred to throughout the sections above.

Feller and Fitzgerald (2002) examined hundreds of OS application projects from Cosource (cosource.com, 2001), Freshmeat, and the Red Hat Linux 6.0 distribution. They categorized this OS software based on functional area (shown in Table 2.1).

Table 2.1 - OS project categorization by Feller and Fitzgerald (2002)

Category	Description	Example
<i>System Environment</i>	Software that supports efficient execution of various application software.	Operating systems, and APIs
<i>Applications</i>	Software designed to help the user to perform singular or multiple related specific tasks. It helps to solve problems in the real world.	Accounting software, office suites, graphics software, and media players
<i>Development</i>	Programs that software developers use to create, debug, maintain, or otherwise support other programs and applications	Eclipse Java IDE, Bugzilla and NetBeans
<i>User Interface</i>	Systems by which users interact with a machine	Eboard (a chess user-interface for internet chess servers)
<i>Documentation</i>	Applications for producing documents	Open Office
<i>Amusement</i>	Software designed for entertainment purposes	Games and media players

Another categorization was provided by Daniel et al. (2009a) who categorized OS projects based on their software complexity and evolution path. They identified six categories of projects: user-centred, controlled, atypical, personal, abandoned, and intractable. They used various dimensions to evaluate OS projects for these categories, such as development activity, survival and usage. These categories are presented in Table 2.2, taken from Daniel et al. (2009a).

In comparing both perspectives for relevance in this work, it is noted that there is evidence that programmer familiarity with the application domain is important in software comprehension (Storey, 2005, Pennington, 1987a, O'Brien et al., 2004). However, a categorization based on software complexity seems to be more significant in this research context. Given that the agenda in this work is information seeking in maintenance (evolution), it is likely that variations in code complexity will impact on this agenda more directly. Hence Daniel et al.'s categorization (2009a), based on code complexity, is likely to have more merit in this research.

This work provides coverage by focusing on the first three categories – user-centred, control and counter-cultural categories – of this classification. This is because our study focuses on maintenance and the last three categories seem to have short maintenance life-spans.

Table 2.2 - OS project categorization by Daniel et al. (2009a)

Category	Main Criteria	Indicator	Webspace Utilization	Holistic Picture
<i>User-Centred</i>	Increase in both size and structural complexity over time.	Successful across both usage and development indicators: - high numbers of downloads and bug report activity - high total number of releases and cvs commits, as well as continued activity after 1 year.	Almost all of these projects provide a rich set of information about the system, targeted toward users, and giving general information about the project.	Appear as a group of projects that live up to the OS software ideal of creating useful, successful software, but seem to follow the pattern of increasing complexity.
<i>Controlled</i>	Has high growth in size while structural complexity decreases or fluctuates.	Quite successful across both usage indicators and development indicators.	Many of these provide detailed information on their website to encourage development, including listing contributors and providing strong guidelines for developers. Overall, they tend to encourage carefully managed development practices.	Appear as a group of projects that follow the classic, ideal type touted by proponents of OS: they create useful, high quality (i.e., low complexity) software.
<i>Counter-Cultural</i>	Fluctuating size and increasing structural complexity describes the evolution of projects in this category.	They have somewhat mixed success indicators: they have the highest average number of developers and most of them continued to be active after the first year. However, they did not attract a very high number of downloads.	All used their web space, similar to the user-centred projects in the kinds of information they provided, with less emphasis on developer-oriented information.	Appear as a set of projects that, while technically open source, do not adhere closely to the cultural norms of OS software. In many ways they appear to behave more like closed development groups.
<i>Personal</i>	Decreasing or fluctuating structural complexity and fluctuating size.	Do not achieve a high degree of success: they have few developers, little activity, relatively few downloads, and most do not survive beyond the first year.	Just over half of these projects use their webspace, and these provide a modest amount of information about the project with little targeted at users or developers.	Appear as a set of projects that are developed to meet personal goals such as self-expression or learning, and once these goals are satisfied, the project falls by the wayside.
<i>Abandoned</i>	Small increases in size accompanied by increases or fluctuations in structural complexity.	Relatively low levels of development activity or user activity.	Provide little information via their websites. Mainly provide information about the project, with little information aimed at users or developers. Half have no website at all	Appear to be another "classic" kind of OS project: that which is abandoned before achieving a notable level of success.
<i>Intractable</i>	Low growth in size and decreases in structural complexity.	Relatively low indicators of success.	Only half of these projects have websites, and these tended to have relatively low amounts of project, developer, and user-oriented information.	It appears that these projects begin with relatively poor design, and some efforts are directed to improving the design. However, for the most part these efforts do not lead to success and relatively few of the projects continue to be active after 1 year.

2.7 Locating this work with other research in the field

To summarize, Figure 2.7 places this work in context. Program comprehension is a core task in software maintenance. The generic program comprehension model consists of a knowledge base (that includes a mental model of the system in question), the external representations available to the programmers and an assimilation process (Good, 1999). According to O'Brien (2007) and Cleary (2007), information seeking plays an important role in supporting the components in a comprehension model. During the comprehension process, information seeking is the external process that involves consultation with external representations to assist in the assimilation process providing input for programmers' knowledge base to develop their mental models. Similar to information seeking concept location also plays an important role in supporting program comprehension. In this context, concept location is a specific sub-task of information seeking – that of locating some target (a user feature or more general concept) in the system. Hence, it can be contextualized as a (location) goal of information seeking.

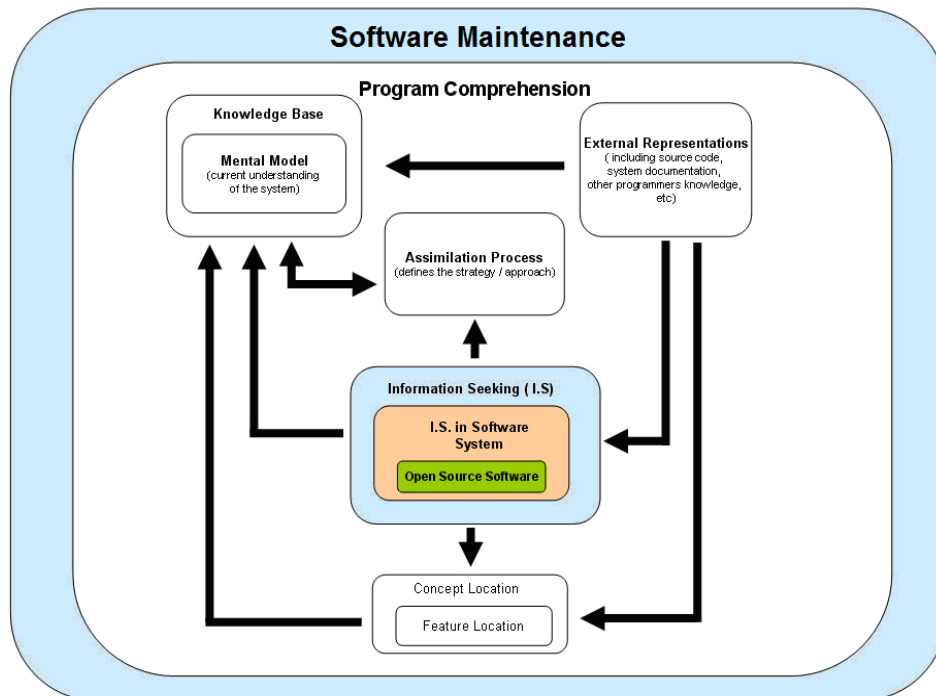


Figure 2.7- Information seeking in OS software maintenance

2.7.1 Differences between This Study and Previous Works

Within this research area, O'Brien (2005) and Marcus (2005) have studied the information seeking processes of programmers during the maintenance of commercial software systems. In complementary research, Singer (1998) and Seaman (2002) have studied the information sources that programmers use when seeking information, also in commercial scenarios. The work reported in this thesis extends this research by focusing on delocalized open source development, in the tradition of O'Shea (2006), where the developer mailing lists of OS projects are analysed to explore the programmers' information seeking efforts. As the characterization above shows, OS development is substantially different from commercial software development and findings in one domain need not necessarily apply to the other.

There have been several empirical studies that aim to explore the types of information sought by programmers in the context of software comprehension (Singer et al., 1998; Ko et al., 2007; Letovsky, 1986; Pennington, 1987; Good, 1999; Wiedenback et al., 1991; O'Shea, 2006; Buckley et al., 2004). These studies focus on the information that programmers need and the information that they find difficult to obtain during software maintenance, thus potentially informing the design of software tools.

However, most of these studies are derived from the existing "information type" schema developed by Pennington (1987). As this schema was developed through a theoretical review of the information available to individuals in small segments of code, it is possible that it ignores other artifacts produced by the development team and that it ignores some information seeking requirements specific to larger code bases and team-based development (Sharif and Buckley, 2009a). An illustrative example is the location information type identified by O'Shea (2007). Late in her thesis research, O'Shea (2007) empirically established that programmers sought the location of a specific piece of code within the software system. While this finding was in line with feature and concept location work, O'Shea attributed the lateness of this finding to the adoption of

Pennington's schema. This "theoretical harness" thus potentially constrained O'Shea's work and has the same potentially constraining possibilities for this body of research.

In contrast, Ko et al. (2007) observed programmers while they were working in-vivo with proprietary or commercial software development teams and identified the information that they sought through the observations, in an open coding fashion. Similarly, Sillito et al (Sillito et al., 2008, Sillito et al., 2006) also used open coding protocol to observe programmers while they are performing change task. Their study was done in two different settings: a laboratory and an industrial setting and characterized the information required by programmers while they are working with code.

However, the work reported in this thesis differs from Ko et al. (2007)'s and Sillito et al (2008, 2006)'s in term of focus. While Sillito et al (2008, 2006) limited their observation to focus only on the very specific activity that is change task, this thesis reports on a broader context and wider range of programmer's activities similar to Ko et al. (2007)'s study. In contrast with Ko et al. (2007)'s study (and indeed Sillito et al (Sillito et al., 2008, Sillito et al., 2006)) that observed co-located software development teams, this study observed distributed open source programmers.

The work reported here does mirror Ko et al. (2007)'s and Sillito et al (Sillito et al., 2008, Sillito et al., 2006)'s approach in that it relies on a schema directly derived from observations of the information types that programmers seek in-vivo. This frees it from any potentially constraining theoretical harnesses. Instead, it places no restrictions on the information source to derive a holistic information seeking schema. In the next chapter, the methodology for this research is discussed in depth.

Chapter 3

**Methodology and
Implementation**

CHAPTER 3 Methodology and Implementation

3.1 Introduction

In order to study the information seeking behaviour of OS programmers during software maintenance, this chapter reviews the potential empirical methods that can be used. It suggests a mix of qualitative and quantitative methods as the best approach to study programmers' information seeking behaviour.

3.2 Qualitative and Quantitative Approaches

Most authors categorize empirical work based on its potential to utilize numerical data (Dennis Howit, 2008, Oates, 2006). Oates (2006) states that "quantitative" refers to empirical studies that generate findings based on numbers. On the other hand, the term "qualitative" is used for empirical studies that generate non-numeric data, such as words, images, sounds and so on (Oates, 2006). Typically, quantitative data analysis employs statistics, generalization and conclusions. Meanwhile, qualitative data analysis emphasizes deep interpretation of phenomena through non-numeric data such as utterances, images and texts, abstracting them into themes and patterns that relate to the researcher's research topic. However, qualitative researchers are also often able to perform quantitative analysis on qualitative data (Oates (2006).

3.2.1 Quantitative Research

Typical examples of data collection techniques in this paradigm include questionnaires, survey (such as market research) and controlled experimentation.

In the case of controlled experiments, quantitative research can be seen as research that evaluates theories by utilizing statistical analyses on numerical data obtained from observations (O'Brien, 2007). The application of statistical

analysis allows interpretation for generality, allowing researchers to share results in a standardized and accepted format (Neill, 2007).

To raise confidence in findings across a population in such a study, quantitative studies use considerably more participants than qualitative studies and typically employ sampling methods in selecting the participants (Howitt and Cramer, 2008).

One important criterion in sample selection for quantitative studies is consistency. The need for a high level of consistency derives from the “controlled” nature of this approach (O'Brien, 2007), as shown in Golafshani's (2003) suggestion that all participants in a quantitative study should have similar “standardized” conditions. In this context, “conditions” implicitly refers to the participants being standardized as well. For example, in the context of quantitative software comprehension studies, short segments of code were often artificially created to observe programmers' behaviour in a situation where the participants had an equal level of familiarity with that code (i.e. no previous exposure). Likewise, in Good's (1999) experiment, to maintain the equal level of familiarity with code in general, all 74 participants were first year undergraduate students. The students were asked to comprehend short programming codes (6 to 12 lines) by answering comprehension questions. While this heightened the standardized nature of the conditions, it limited the relevance of the work to in-vivo software evolution, as that code was of a different scale and atypically the programmers were entirely unfamiliar with it or its conventions (Von Mayrhauser and Vans, 1995). The limitations of quantitative research include the following:

- 1 Since quantitative research is only focused on measurable aspects of the research topic, it is possible that it misses important contextual (non-quantitative) information with respect to the research topic. As a consequence, rich explanatory qualitative data to determine the “why” of research findings (such as why certain hypotheses failed) may be lacking (Howitt and Cramer, 2008).

- 2 As discussed above, the controlled experimentation in most quantitative studies impacts on the naturalistic nature of the result. This approach often results in a highly artificial context for the participant that possibly limits the applicability of the findings in real situations (O'Brien, 2007).

3.2.2 Qualitative Research

In contrast to quantitative study, qualitative studies do not count or measure, but allow the inclusion of just about any explicit or implicit human communication medium as its dataset: written, audio, visual, behavioural, symbolic or cultural effect (Gibbs, 2009). Common qualitative data mechanisms include in-depth interviews, participant observation and focus groups (Howitt and Cramer, 2008).

Bogdan and Biklen (1998) define qualitative data analysis as “working with (this) data, organizing it, breaking it into manageable units, synthesizing it, searching for patterns, discovering what is important and what is to be learned, and deciding what you will tell others”. Its aim is to attain a complete, detailed, rich and precise description of situations (Denzin and Lincoln, 2000).

Based on the rich datasets it generates, qualitative research is concerned with the social aspects of our world (Hancock, 1998) and aims to address questions concerning:

- Why people behave the way they do;
- How opinions and attitudes are formed;
- How people are affected by the events that go on around them;
- How and why cultures have developed in the way they have;
- The differences between social groups.

According to Westbrook (1994) and Neill (2007), the main uses of this approach are hypothesis formation, explanation and description of phenomena, understanding complex situations, and addressing research areas where scientific exploration is still at a lower level of maturity.

However, there are also limitations in the qualitative approach. The “less formal” nature of this approach makes it harder to interpret results compared to the quantitative approach. The data obtained in this approach is in a variety of forms, of a very large quantity and consists of very rich content. Working with this kind of data can be tedious and time consuming (Black and Rabins, 2006) and there is the risk that the interpretation is biased in accordance with the researcher’s interest (Johnson and Onwuegbuzie, 2004). Another factor that makes it harder to interpret is the absence of statistical analysis which, as stated above, is an analysis technique accepted and easily understood by the research community.

Like quantitative research, the findings of a qualitative study might not generalize to a larger population. This is because of the typically small sample size, the non-random selection of participants and its focus on probing the individual differences amongst its population in depth.

Given that qualitative research addresses complex situations where scientific exploration is still at a lesser level of maturity, qualitative methods would seem to be an appropriate vehicle of analysis for studying information seeking in OS software development. Indeed, Finkelstein and Kramer (2000) explicitly support this when they state that there is growing interest in employing the qualitative approach in investigating software processes in general, for these reasons.

3.2.3 Comparison between Qualitative and Quantitative Methods

As a comparison, it can be concluded that each of the paradigms employs a different approach and is concerned with different perspective of knowledge acquisition, as summarized in Table 3.1. Indeed, Reid and Gough (2000) argue that each paradigm is involved in “seeking a different type of knowledge”. The quantitative method seems to emphasize “confirmation” information about a specific research topic or hypothesis. On the other hand, qualitative methods are likely to emphasize “why” information, reasoning on the research topic or research findings. Hence, comparing them to decide which one is better or worse

is the wrong agenda and unnecessary (Spindler, 1982). Rather, the agenda should be to determine which one is appropriate for the research needs.

This different “type of knowledge” is reflected in a major difference between the quantitative and qualitative approaches: quantitative research is predominantly deductive while qualitative research is inductive (Glaser and Strauss, 1968). Specifically, quantitative research requires a hypothesis before the research starts. The hypothesis will drive the study in that the generated data will be used to evaluate the veracity of the hypothesis. On the other hand, in qualitative research the hypothesis is often derived from the data. Hence, several authors (such as (Snyder, 1995, Johnson and Onwuegbuzie, 2004, Jick, 1979, Dures et al., 2011)) suggest the need for mixing qualitative and quantitative methods. Johnson and Onwuegbuzie (2004) regard this mixed method as a “creative form” of research that allows the researcher to use multiple approaches to answer research questions in a form of triangulation. In addition, a mix of the paradigms allows for confirmation or repudiation of a research question while also determining the reason for the findings. That is, quantitative findings (figure-based) might lead to surprising results and qualitative data can be used to derive an explanation for these findings.

Table 3.1- Summary of main features of qualitative and quantitative research, derived from (Neill, 2007)

	Qualitative Methods	Quantitative Methods
Aim	Aim for a complete, detailed description (“why” information)	Aim to figure out “what” is observed and predict the future based on constructed numerical / statistical model
Maturity Of The Discipline	Recommended for earlier phases of research exploration	Recommended for later phases of research for highly focused experiments
Researcher Agenda	Researcher may only know roughly in advance what he/she is looking for	Researcher knows clearly in advance what he/she is looking for
Evolution Of Empirical Design	The design emerges as the study unfolds	All aspects of the study are carefully designed before data is collected to ensure control
Data Gathering Instrument	Recording devices, video cameras, interview, in-vivo observation etc.	Questionnaires, quizzes, experiments etc, to collect numerical data.
Data Format	Words, pictures or objects (rich and precise)	Numbers (numerical data)
Data Analysis	More 'rich', time consuming analysis, less able to be generalized	Comparatively easier to analyze, but may miss contextual detail
Objectivity/ Subjectivity	Researcher tends to become subjectively immersed in the subject matter.	Researcher want to remain objectively separated from the subject matter

Hence, the most plausible approaches are those that offer multiple, independent, but converging strands of research (Perry et al., 1997), possibly from both research methods (mixed). In other words, these two research methods are complementing each other in answering research questions. This is the approach proposed in this research.

It is in contrast with typical computer science research practice that normally tended to use quantitative approaches, as shown in Tables 3.2 and 3.3. These tables report on two leading forums for empirical studies in computer science: Empirical Studies of Programmers (ESP, discontinued) and the International Journal of Empirical Software Engineering (ESE). They show the predominance of quantitative studies and the lesser number of studies with (at least) a qualitative element. However, there are an increasing number of studies that employ both qualitative and quantitative research methods, thus supporting the approach adopted here: These studies benefit from having two streams of orthogonal data that serve to complement each other, as outlined above.

Table 3.2 - Quantitative and qualitative studies in ESP (Buckley, 2002)

ESP	No. of quantitative studies reported on	No. of qualitative studies reported on	No. of studies with elements of both
1986	11	3	3
1987	6	4	5
1991	9	4	3
1993	4	2	5
1996	10	0	7
1997	8	1	4
Total	48	14	27

Table 3.3 - Quantitative and qualitative studies in ESE⁵

ESE	No. of quantitative studies reported on	No. of qualitative studies reported on	No. of studies with elements of both
2000	8	0	0
2001	3	2	4
2002	9	0	2
2003	8	2	1
2004	4	1	4
2005	10	4	3
2006	11	3	3
2007	16	3	5
2008	17	1	5
2009	17	1	5
2010	17	1	5
2011	13	6	6
TOTAL	133	24	43

In conclusion, the reason for using a qualitative and associated quantitative approach for this research agenda may be summarized as follows:

- This research is exploratory in nature, that is, it is concerned with theory building and generation rather than the testing of hypotheses – thus, the need for qualitative methods.
- Qualitative methods more intensively engage with the participant and their environment, typically heightening ecological validity.

⁵ Result of ESE 2000 to ESE 2006 were adopted from O'Brien (2007).

- While the qualitative methods pay more attention to detail and interpretation than quantitative methods, quantitative methods allow the researcher to summarize the data and thus see trends, not immediately apparent in the rich detail.

3.3 Data Analysis Methods

Given the conclusions in section 3.2, this research adopted aspects of the Grounded Theory approach for qualitative analysis along with content analysis for subsequent quantitative analysis. These techniques, along with their rationale for inclusion, are now described in detail.

3.3.1 Grounded Theory

Grounded Theory is a process in which researchers “bring up” theory that resides (grounded) in the data (Strauss and Corbin, 1998) and is commonly used in the qualitative paradigm. The data in its various forms is typically obtained from the real environment (for example, real programmers working on real software projects). The data is then analyzed by employing a coding procedure to illuminate patterns or concepts that, in turn, generate theories (Strauss and Corbin, 1998). This process is repeated until it reveals no new pattern or concept. Charmaz (2006) regards this procedure as a systematic analysis of the data and states that “the rigor of Grounded Theory approaches offers qualitative researchers a set of clear guidelines from which to build explanatory frameworks that specify relationships among concepts”. Kelsey (2003) described the detailed process in Grounded Theory as a “data dance” and presented it in a framework reproduced in Figure 3.1.

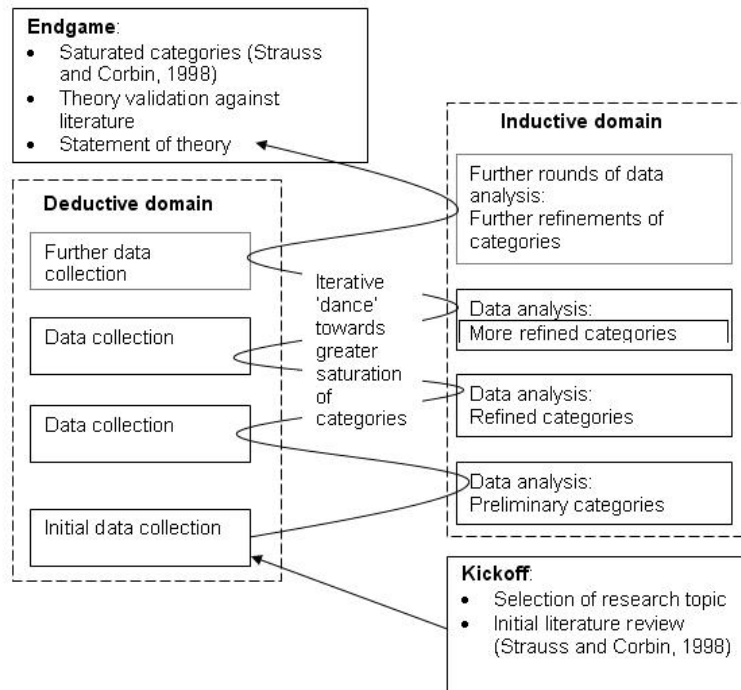


Figure 3.1 - The Grounded Theory “data dance” (Kelsey, 2003)

According to Kelsey (2003), the Grounded Theory process is formed on an inductive and deductive cycle. As illustrated in Figure 3.1, the process is a data collection and analysis process where each round of data collection is based (deductively) on findings obtained from the preceding round of data analysis (inductively). This process is repeated iteratively, until the theory becomes “saturated” (Strauss and Corbin, 1998). Saturation is the means of identifying when the appropriate sample size for the study (see Section 3.5) has been obtained. It maps to Strauss and Corbin’s (1998) concept of “theoretical saturation”. Theoretical saturation occurs when:

- No new or relevant data seems to emerge regarding a category or categories.
- The category is well developed in terms of its properties and dimensions, demonstrating variation.
- The relationships among categories are well established and validated.

In other words, in achieving saturation, the researcher has to repeat the process (of data collection and coding) and consequently expand the sample size until it discovers no new data that enhances the theory. Power (Power, 2009), in agreement, states that theoretical saturation is about focusing on interesting findings from the data and finding more evidence until no new insights emerge. Power also explains that there is no “magic” number of samples to satisfy the saturation criteria.

Figure 3.2 expands on the process of building Grounded Theory. As shown in the figure, the process consists of five phases: i) deciding on the research problem; ii) framing the research question; iii) collecting data and theoretical sampling; iv) coding and analysing the data; and v) developing the theory.

The “frame research question” phase differentiates Grounded Theory from the quantitative approach. Grounded Theory does not validate hypotheses that are created during earlier stages of the research. Instead, it starts with a very general research question. Then, any information (found during the coding / analysis process) that is relevant to this research question is used to construct (emerge) the theory (Bitsch, 2005). This is how Grounded Theory determines the scope of the “phenomenon to be studied” (theoretical sampling) during the next cycle. Figure 3.2 also shows that, in the event that saturation is not yet achieved in the cycle, the researcher will revise the coding process on the existing data, or if necessary, revise the data collection process and revise the results of the recent theoretical sampling.

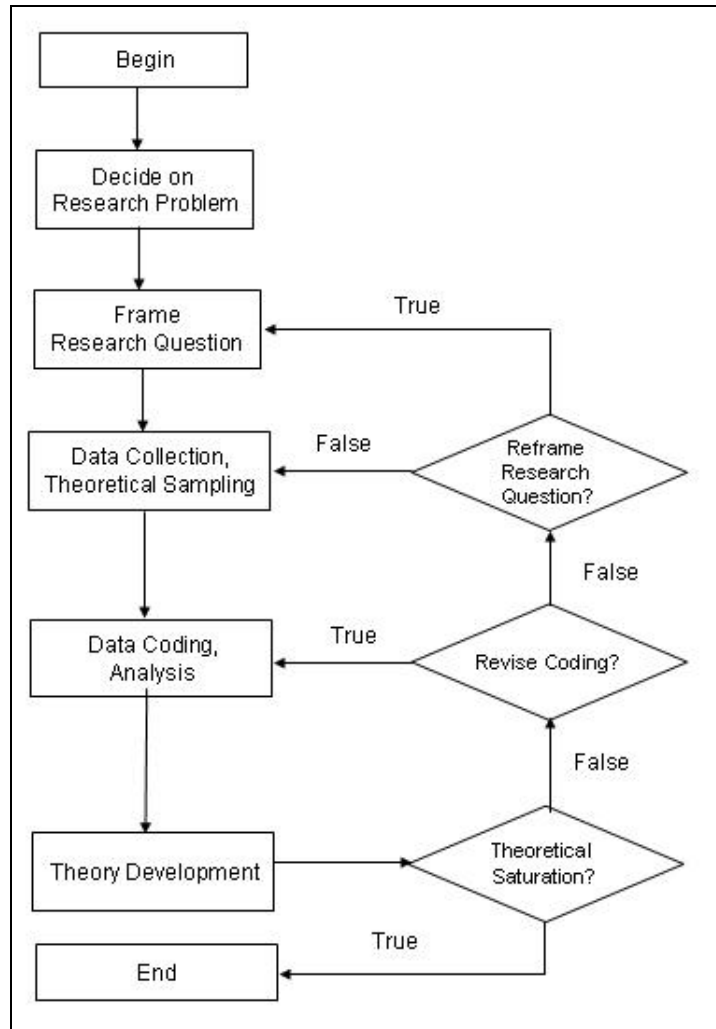


Figure 3.2 - Grounded Theory flowchart (Bitsch, 2005)

During the coding and analysis phase, repeated themes are identified and coded as concepts. Strauss and Corbin (1998) proposed a basic coding sequence within this phase to construct the emerged theory. According to them, the analysis should follow the basic coding sequence illustrated in Figure 3.3 (open coding → axial coding → selective coding). The coder should thus iterate between primary data and the emerging theoretical framework. Charmaz (2006) regards the initial coding phase as a process of categorizing segments of data with a short name that “simultaneously summarizes and accounts for each piece of data”(Charmaz, 2006). In other words, the emerged codes and categories are directly derived from the data.

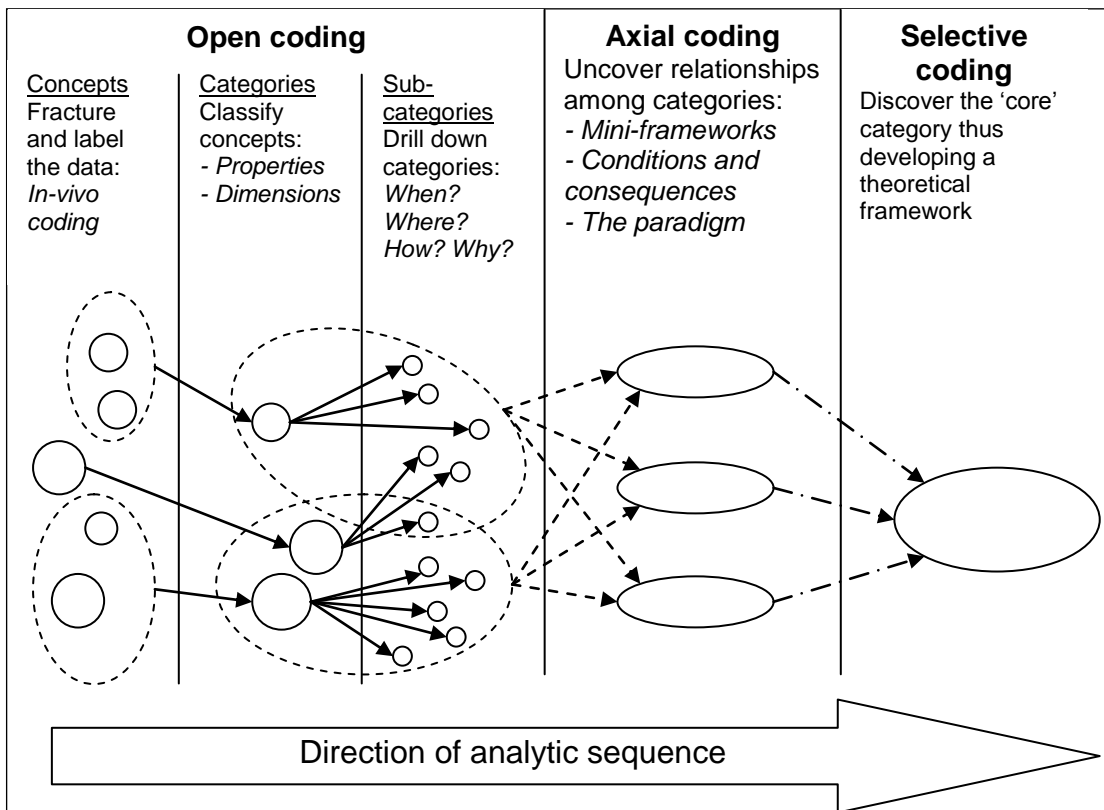


Figure 3.3 - The Grounded Theory analytic process (adapted from Harwood, 2002)

i) Open Coding

Open (emergent) coding is relatively free from constraints and subject only to whatever patterns are emerging from the data (Power, 2002). In this thesis, open coding is used to identify categories of questions in OS mailing lists. This is called inductive data analysis in open coding (Hoepfl, 1997). Data is compared and similar incidents are grouped together and given the same conceptual label if appropriately close. The process of grouping concepts at a higher, more abstract level is called categorizing (Pandit, 1996). The goal here is to create descriptive, multi-dimensional categories which form a preliminary framework for analysis as suggested by Hoepfl (1997). Then, similar data is correspondingly coded based on the preliminary framework. Figure 3.4 illustrates the open coding sequence.

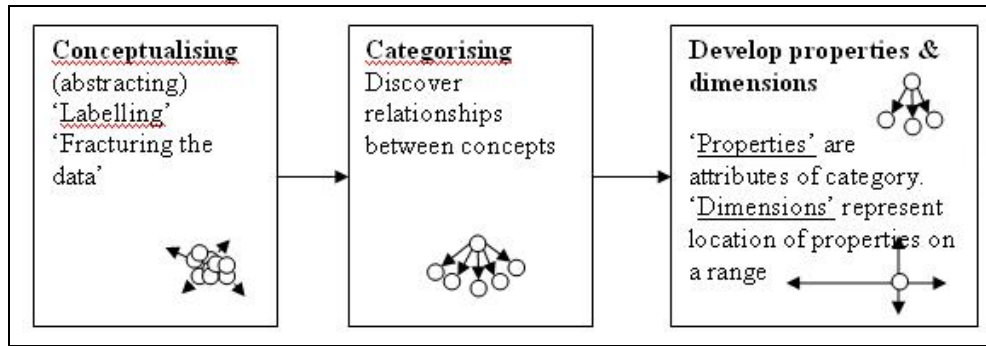


Figure 3.4 - Open coding sequence according to Strauss and Corbin (1998)

The first step is conceptualizing. It is done by labelling the data in the dataset. A label could be any insight into the data that is possibly related to the study. The next step is categorization, when the data is categorized based on the relationships between the identified concepts in the data. For example, all questions (found in the mailing list) that have a similar topic could be categorized under the same category. Finally, analysis is done on each category to identify and contextualize possible dimensions of attributes in each category.

ii) Axial Coding

The purpose of axial coding is to relate categories to sub-categories (Charmaz, 2006). Thus, axial coding allows the development of major categories, although they may be in the early stages of their development (Charmaz, 2006). It does this through specification of the properties and dimensions of the categories. Hence, Strauss (1987) views axial coding as building “a dense texture” of relationships around the “axis” of a category.

iii) Selective Coding

Selective coding, also called focused coding, is more directed, selective and conceptual than open coding and axial coding (Charmaz, 2006). It is done after the researcher establishes some strong analytic directions from previous coding to synthesize and explain larger segments of the data. “Selective”

means using the most significant and/or frequent earlier codes to sift through a large amount of data. One goal of this process is to determine the adequacy of those codes.

According to Charmaz (2006), various qualitative data mechanisms such as in-depth interviews and focus-group observation can be used with the Grounded Theory approach. While it is noted that in-depth interview would also be useful for this study, the delocalized nature of open source programmers make an interview approach impossible. Alternatively, there is a suggestion that email could be used as a medium to interview OS programmers that participated in the dataset (to know more about their information requests, as found in the dataset). However, given the fact that the conversations taken from the mailing list archives were several years old in many cases, an email interview seems unrealistic. Hence, group observation through the mailing lists in isolation seems to be more realistic in the context of this study.

In conclusion, Grounded Theory offers an inductive approach that can cast off the theoretical harness of other works in this area, yet is systematic in its application. Hence, this work largely concentrates on the open coding aspect of Grounded Theory, deriving some sub-categories through axial coding.

3.3.2 Content Analysis

Holsti (1969) suggests a very broad definition of content analysis when he defines content analysis as "any technique for making inferences by objectively and systematically identifying specified characteristics of messages". However, perhaps the most widely accepted definition of content analysis is that found in Krippendorff (2004):

Content analysis is a research technique for making replicable and valid inferences from texts (or other meaningful matter) to the contexts of their use.

During content analysis, attempts are made to systematically and reliably analyze data in a repeatable fashion to identify overall trends and to heighten confidence in the resultant theory. The analysis usually results in reducing data into more concise units/categories of information (Krippendorff, 1980).

The content analysis method has been used widely in many disciplines including anthropology (Toomere, 2002), ethnography (Altheide, 1987) (Eltinge and Roberts, 1993), literature (Julien, 1996), political science (Budge and Klingemann, 2001) psychology (Peterson et al., 1985) and software engineering (O'Shea, 2006, O'Brien, 2007, Good, 1999, Corritore and Wiedenbeck, 2000, Kelly and Buckley, 2009). In fact, a workshop on content analysis ("Applications and Emerging Trends in Software Engineering Practice") was held as part of the Conference of Software Technology and Engineering Practice 2004, illustrating its acceptance as an empirical method in software engineering research.

The content that is analyzed can be in any form to begin with, but before content analysis can begin, the data needs to be preserved (transcribed) in a written form that can be analyzed. In the context of this thesis, the data from the mailing list is already in written form but information requests need to be extracted.

The transcribed data (in this study, the extracted questions) is typically segmented and each segment is analyzed and categorized. Most content analysis uses a scheme of categories that are relevant to the research objectives.

The next step is coding and classifying the data (categorization). In doing this, Krippendorff (1980) provides standard guidelines for content analysis:

1. Select the size of the data elements to be analysed; that is, decide how big a chunk of data is analysed at any given time (a line, a sentence, a phrase, a paragraph). It is imperative that whatever is decided is carried through consistently during the whole process.
2. Decide upon the units of meaning (the categories that the data will be coded into). Categories must be inclusive (where all utterances fit a

category), although this is often achieved by means of a “bucket” category.

3. Define the categories precisely (i.e., what are their identifying properties).

With a-priori coding, the categories are established prior to the analysis based upon some theory (Good, 1999). The coding in this study is the open coding derived from the data through Grounded Theory. In addressing points 2 and particularly 3 above, several researchers, such as Good (1999), O’Brien (2007) and O’Shea (2006) have drafted a coding manual to allow other researchers to evaluate their coding schema, refine it and perform the same coding on the same and different datasets. Such coding manuals are documents containing explicit descriptions of the categories along with how to identify these categories in the data. In the case of a-priori coding schemas, these manuals can be defined in advance and used in conjunction with multiple coders to establish reliability. With an open coding protocol, manuals like this can only be generated as or after the data is traversed in a Grounded Theory fashion, resulting in an explicit statement of the coding protocol undertaken, and making the protocol for analysis transparent.

Advantages of the content analysis method include:

- It can be less intrusive in nature. Content analysis can be applied to documents already created, recordings of in-vivo meetings or, in the OS software maintenance context, communication emails of OS programmers (O’Shea and Exton, 2004).
- It can be applied to a wide variety of empirical data. That is, it can handle unstructured matter as data (Krippendorff, 2004) (as long as it can be segmented).
- The data is often “contextual”, and so content analysis can be context sensitive. For example, consider OS software maintenance; it is a realistic work scenario that will probably allow the gathering of ecologically valid

contextual data. Therefore, it allows the researcher to process data texts that are significant, meaningful, informative, and even representative to others (Krippendorff, 2004).

- It has the ability to quantify qualitative data (O'Brien, 2007), through counts of different categories.

3.3.3 Other Methods of Textual Analysis

There are alternative methods for exploring texts systematically, such as discourse analysis (Tannen, 2001), conversation analysis (Krippendorff, 2004) and rhetorical analysis (Krippendorff, 2004). These methods of analysis share similar characteristics with content analysis in that they: (1) require a close reading of relatively small amounts of textual matter, and (2) involve the interpretation of given texts into new narratives. However, these alternative methods were not considered to be appropriate for this research. This is because these techniques are more directed to the study of the structure and sequence of language and conversations (Tannen, 2001, Marshall, 1998) rather than focusing on the content of utterances (in this case, content such as the information that programmers seek along with their frustrations in seeking this information).

Another textual analysis method for qualitative research is narrative analysis. In this approach, the subject matter is analyzed as narrative. Narrative analysis can be used to complement content analysis (Smith, 2000). According to Riessman (1994), narrative analysis “permits a holistic approach to discourse that preserves context and particularity”. This approach is possibly appropriate in building on this work to analyze dialogues in email such as analysis of the response period, response rate, depth of response and ultimate outcome.

3.4 Method of Data Analysis Employed

Here the method of data analysis employed in this research is discussed. It is a combination of aspects of Grounded Theory analysis and content analysis. This is presented in Figure 3.5.

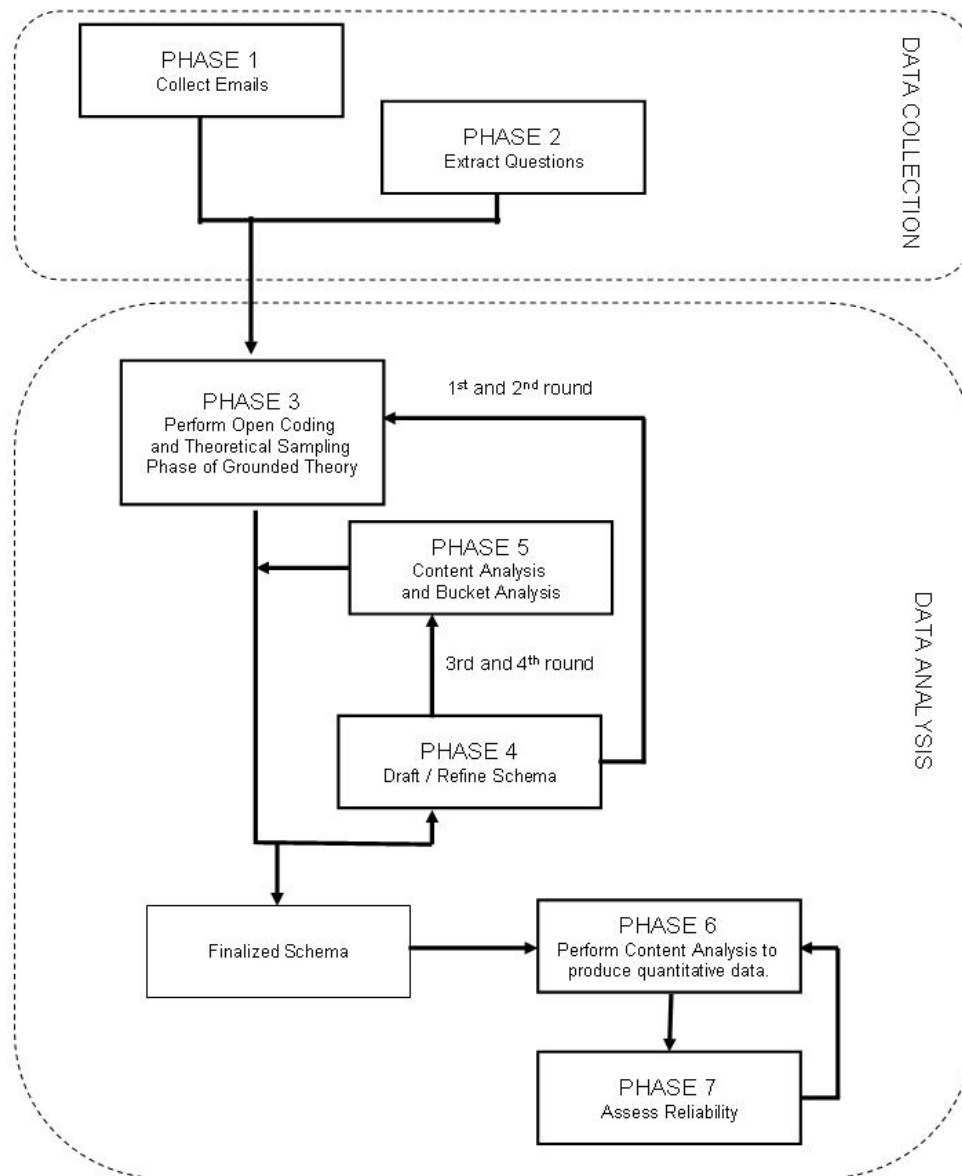


Figure 3.5 - Seven-phase model for data collection and analysis

Grounded Theory and content analysis will be used in the creation of a schema of “information seeking concerns of OS programmers” on the dataset. During the first and 2nd round, Grounded Theory will be used in the creation and refinement of the initial schema. With this schema partially established, further rounds of

analysis (the 3rd and 4th rounds) will be performed on an extended version of the dataset, using this preliminary schema and content analysis. This will analyze the prevalence of the preliminary categories and exceptions that arose to these categories, resulted in new categories or refinement of the existing categories.

In the interest of scoping this research, the relationships between the information seeking concerns were only partially analysed. While it is acknowledged that this is important work, a more complete exposition is left as future work. Consequently, this work concentrated primarily on the open coding of the data and theoretical sampling of the data with respect to the derived categories. Some hierarchical relationships were identified as the work progressed, in line with axial coding practice but the work did not emphasize this aspect of theory building.

In forming an initial information seeking schema, data in the form of emails was gathered from two OS developer mailing lists consisting of 288 emails. Later, 98 questions were extracted from those emails and analyzed for potential categories. 708 questions were subsequently captured from 2104 emails taken from 17 mailing list archives of OS software projects. The resultant schema was iteratively refined in the tradition of Grounded Theory (Pandit, 1996), until it became saturated (Howitt and Cramer, 2008) with respect to the derived categories over the course of these 17 mailing lists archive.

This resulted in the emergence of categories of information seeking from the data. Afterwards, using these categories and the properties in the text that reflected these categories, a scheme (that we called the information seeking schema) was developed and content analysis was carried out to find overall trends in the data. The findings were related back to other information seeking schema in the literature for comparison and analysis purposes.

Hence, the theory was, in effect, derived and refined in a data-driven fashion. The method of data collection, the coding process and the schema development are now described in more detail.

3.5 Sample Size and Validity

According to Strauss and Corbin (1998), the key to success in using Grounded Theory is to capture enough in-depth data to reveal patterns, concepts, categories, properties, and dimensions of the given phenomena. They called this “theoretical saturation”. Hence, appropriate sample size and type is important to get saturated data. As stated in Section 3.3.1, Power (2009) points out that there is no magic number as a guide to when the data is saturated. Power does, however, suggest two guidelines for saturation justification which were employed in this study:

- Wide sampling – the coverage of the analysis should be maximized. In this study, different OS software project categories were assessed based on the OS project categorization by Daniel et al. (2009a). Their categorization of OS software projects was based on the software complexity and evolution path. In addition, different stages of evolution in OS software were studied by analysing mail archives that were taken from different phases in different software systems’ evolution.
- Theoretical sampling – further sampling should be done based on any emerging issues (theory) that arise during the analysis. This includes reappraising the categories and deriving alternative categorizations, evaluating the significance of not well-supported categories and deep investigation for certain prevalent categories, with strong emphasis on the dataset. This is in line with Charmaz’s (2006) approach of “coding incident to incident” to identify the properties of emerging concepts.

3.6 Method of Data Collection

Data collection was done in three phases as shown below:

1. Identification of the mailing list archives

Mailing list archives are identified based on several criteria such as project type (Daniel et al., 2009a), number of emails in the archive, software evolution stage (when the emails in the archive were created) and length of the archive. Saturation criteria drove the lists chosen, so most of the mailing lists were identified in a staggered fashion, as the data analysis proceeded.

2. Extraction of the data from the mailing lists

Initial investigations showed that a considerable number (approximately 20%) of the questions in programmers' emails were asked without explicit indicators like question marks or signalling words such as "what" and "where" (Sharif and Buckley, 2008b, Sharif and Buckley, 2009a). As a result, the questions in the mailing list had to be extracted manually. Hence, a coder read the entire dataset and identified all the questions, the responses generated and the period of time between the questions and their first and last gained response.

3. Preparation for analysis

All of the questions found in the emails were then isolated (in this case, in an MS Excel spreadsheet). Categorization and content analysis could then be applied, initially to produce a schema, and subsequently to produce the quantitative data recorded in the Excel spreadsheet. In several cases question sentences contained more than one question (such as: "Can you tell me how and why this error happened?"). In such instances, the questions were duplicated and highlighted (bolded) according to each sub-question found in the sentence.

3.7 Research Ethic

This study considers the need to maintain an ethical approach in observing human behaviour. This kind of observation, typically involving “prolonged immersion” in the daily activities of the observed community, would suggest the need of informed consent (Moore and Savage, 2002). They state that informed consent is required so that the participant is aware they are being studied, understands the purpose of the study and understands how data will be guarded and used. Informed consent is also important, to get participants’ agreement to be observed.

However, given the fact that this study is done in an Open Source context the situation is less clear-cut. There are some authors who suggest the need for guideline regarding ethics in studying OS communities (El-Emam, 2001, David M. Berry, 2004) especially for OS groups that put access restrictions that only allow registered members. However, none of the developer mailing lists in this study had these restrictions.

As such the mailing list is publicly published material and thus is available for public viewing and presumably analysis. Hence, it seems that the need for informed consent is diminished in this instance. In addition, David M. Berry (2004) argues the difficulties and problems with getting consent from OS communities. He argues that, the delocalized nature of OS programmers, as well as the diffusion and shifting of OS programmer communities over time would make seeking consent extremely difficult.

3.8 Coding Process and Analysis Schema Development

During the initial coding, an open coding procedure was carried out without the aid of a coding manual or schema; the coder effectively created the categories entirely. Accordingly, the coder immersed himself in the transcript data, seeking to gain as many insights as possible into the information seeking behaviour of the

programmers, and began to create categories based on the contents of portions of the transcripts (the questions) being examined, as suggested by Pandit (1996) and O'Brien et al. (2001).

The analysis was performed iteratively, with each iteration marked by a preliminary content analysis to identify prevalent and infrequent categories. Each iteration was also marked by several discussions/reviews with the research supervisor. These discussions were based on an exercise where the author and research supervisor randomly chose 30 questions from the sample and applied the schema independently. The meetings then compared results to assess the schema's reliability and identify points of issue.

Discussions with the research supervisor were also held to mediate possible problems related with the main researcher's background as a non-native English speaker (see section 1.8). On top of the discussions mentioned above, discussion were also held to resolve these issues:

- Ambiguity of question sentences that made it difficult for the main researcher to perform categorization.
- Defining the relationships among concepts. This is related to the Axial Coding stage (see section 3.3.1)
- Reflections on findings with respect to the literature in the related area.

It should also be noted that the main researcher's background (discussed in section 1.8) would give him advantages in the coding process. His work experience as a programmer would help him as he tried to identify concepts embedded in the data. It could be argued that his lack of Java experience would be a problem in this regard. However, his experience in C and C++ more than compensated for his lack of Java experience.

Over time, a number of provisional categories began to emerge over these iterations. These categories emerged, not just by analysing the sentences in isolation, but often by determining the semantics of the immediate context of the questions identified. That is, the text surrounding the questions was often

analysed to determine a truer meaning for the question. The emergent categories were then applied to other question datasets and refined by means of merging and renaming categories. The coding process was repeated again.

Then, as a preliminary schema was established, the coder performed content analysis on new datasets, identifying instances where the questions fell outside the preliminary categories.

These questions were used to derive new categories of information seeking. Finally, the set of categories seemed increasingly resistant to change or, in Grounded Theory terminology, they were saturated.

Once the schema was finalized in this fashion it was used as the basis for a coding manual for content analysis, identifying the themes (found during open coding) in the dataset. Content analysis was used as a quantitative approach to measure the prevalence of emergent categories in the complete dataset.

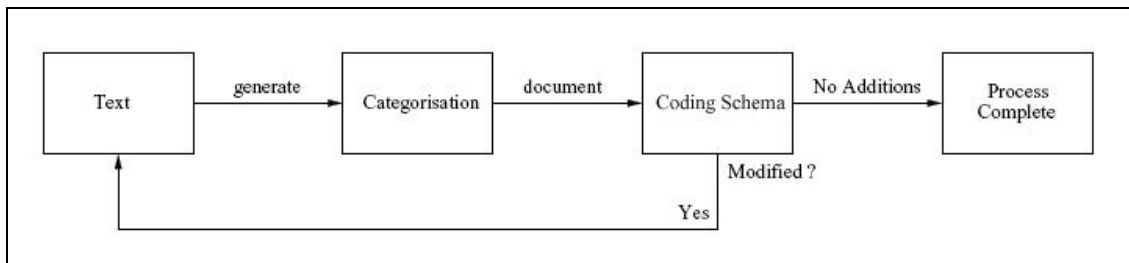


Figure 3.6 - Process of category creation

3.9 Conclusion

This chapter reviewed the empirical methods that were used to study programmers' information seeking behaviour: a derived approach based on aspects of Grounded Theory, aligned with content analysis. This approach frees the study from the theoretical harness placed on other work in the area, while also allowing the findings to be presented succinctly and to be probed for trends in a systematic fashion. The remaining chapters describe the derivation of the schema, and the results made available through the application of this schema.

Chapter 4

Schema Development

CHAPTER 4 Schema Development

4.1 Introduction

In this research we derive a schema of OS programmers information seeking that is used to answer some of the research questions in this thesis (see section 1.5) and can also be used in the future as a framework to provide guidance for further research work in this area.

This chapter presents the process of schema development. It starts by briefly summarizing the iterative protocol used to derive the schema and then steps through each of these iterations detailing the refinements made.

4.2 The Analysis Process

All the datasets were taken from Jakarta project website⁶ due to the fact that it is home to the major open source Java projects (O'Shea, 2006). Java is well known as the most widely used programming language in OS projects (Karus and Gall, 2011). More generally, Java is also acknowledged as one of the most popular programming languages in general (Chen and Dagnat, 2011) and one of the most frequently used object oriented languages (Anik and Baykoç, 2011). In contrast to the fact that most of the studies in this area were influenced by Penington's (1987a) study which looked that the information available in small procedural code segments, the Java language was thought to be more representative of real, OS, object-oriented programmers' scenarios.

Initially, a scraping program that would automatically extract questions from the mailing list was proposed. The program would identify questions appearing in mailing lists based on question marks or question-signalling words such as

⁶ <http://jakarta.apache.org/>

“what” and “where” before extracting the associated questions and compiling them as a dataset for this study. However, in our initial investigations we found that a considerable number (approximately 20%) of the questions contained in programmers’ emails were asked without explicit indicators such as question marks or signal words (Sharif and Buckley, 2008b, Sharif and Buckley, 2009a). As a result, the questions in the mailing list had to be extracted manually.

All of the questions were then individually isolated in a spreadsheet, ready for analysis. Such segmentation is a near-prerequisite for data preparation when analyzing textual data in this fashion (Good, 1999, Shaft, 2001).

The analysis work involved naming and categorizing each of the questions in the spreadsheet. This analysis was carried out iteratively over a number of datasets, as suggested by Grounded Theory. However analysis iterations were also performed within each "dataset" iteration, so we refer to dataset iterations as "observation-cycles" in this thesis.

The researcher carried out the naming and categorizing analysis using an open-coding procedure without the aid of a coding manual or schema. Accordingly, the researcher immersed himself in the transcript data, seeking to gain as many insights as possible into the information seeking behaviour of the programmers, using the questions and the mailing list context.

This open-coding procedure was done iteratively for the first 2 observation-cycles. In each of these iterations, several discussions with another researcher (the research supervisor) were undertaken to review the findings. Prior to these discussions, data was selected randomly by the author, and categorized by both researchers independently. Then the researchers compared results and discussed difficulties, as a basis for refining the schema. Over time, a number of provisional categories began to emerge with respect to the contents of the questions being examined (O'Brien et al., 2001, Pandit, 1996) and the refinements. Those categories were then applied to other question datasets in subsequent observation-cycles and refined, typically by means of adding to, merging and renaming categories. Towards the end of the 4th observation cycle,

the set of categories seemed increasingly resistant to change resulting in the final schema.

The mechanism of schema evolution used in the latter stages of this study is based on the prevalence of each emerged categories. Specifically, question categories with low frequency were considered for exclusion and categories of questions with high frequency were considered core. This latter group was evaluated for possible break up into finer grained categories. Questions that did not seem to fit into the existing schema were assessed for possible schema expansion.

To maximize the coverage (wide sampling) of the schema, the observation-cycles were done on various datasets:

- i) An initial random dataset from Java projects (comparatively small datasets as an initial basis during the pilot study).
- ii) Datasets from different stages of software evolution.
- iii) Large time-scale datasets (96 months).
- iv) Datasets that reflect successful OS project categorizations.

Please note that the examples of questions shown here (chapter 4 and 5) were taken from the actual dataset.

4.3 Pilot study on the Initial Dataset (1st Observation-cycle)

The study described in this section is a pilot study that examined the information sought by open source programmers during software evolution of the Java Bean Scripting Framework (BSF) and Java Development Tool (JDT) project. The JDT is an OS project concerned with enabling Eclipse for Java development. The BSF is an OS project concerned with allowing Java applications to contain embedded languages, through an API to scripting engines. This study was reported in (Sharif and Buckley, 2008a).

The BSF and JDT were taken from the Apache Jakarta project website (Jakarta, 2007). The Apache Jakarta project website was chosen due to the fact that it is home to the major open source Java projects (O'Shea, 2006) during the time of study. These two projects were picked at random, from a sub-set of OS projects that had strongly active developer mailing lists.

4.3.1 The Dataset

The BSF programmers' mailing list (BSF, 2007)⁷ was captured for the period of January to September 2007 (the only available archive in this mailing list for the year at the time of analysis), while for the JDT list (JDT, 2003)⁸, mails were captured from January to December 2003 (the second year of this archive) and these 2 sets served as the data which would be used for open-coding. The resultant dataset consisted of 275 communications between the programmers' community and, from this dataset 145 questions asked by the programmers were extracted. The number of emails and extracted questions for each dataset is shown in Table 4.1 below.

Table 4.1- Dataset for pilot study.

Email Archive	Number of Emails	Extracted Questions
BSF (Jan - Sept. 2007)	128	44
JDT (Jan - Dec. 2003)	147	101
Total	275	145

In an attempt for wider sampling, email archives used as the dataset in this pilot study were chosen based on differing periods of time when the email communication happened. The BSF archive (from January to September 2007) used in this cycle was the latest archive during the study: the sixth year of the project (representing a matured project). In contrast, the JDT archive (January to

⁷ <http://jakarta.apache.org/site/mail2.html>

⁸ <http://dev.eclipse.org/mhonarc/lists/jdtdev/maillist.html>

December 2003) was the most active archive (biggest number of emails) over the project lifespan to date (the second year of the project).

4.3.2 Schema Evolution

This analysis started with a pilot analysis of 44 questions from the BSF archive. The same method was then repeated with the 101 questions from JDT archive. This is in line with the iteration protocol for grounded theory approaches, as suggested by (Kelsey, 2003) and (Bitsch, 2005).

For each OS project, the researcher spent about 2 weeks extracting questions from the mailing list and performing open coding on the data. Reviews with the research supervisor were held at the end of each of these analyses, where the research supervisor was asked to code approximately 30 of the questions in the dataset according to the preliminary schema types. Subsequent discussions were based on comparing the researchers' initial coding efforts as described above. The resultant discussions led to reflections on the schema and refinement of the codes. The reflection process included a reassessment of each emerging category.

This led to 10 provisional types of information need, identified from the BSF archive, as shown in Table 4.2 below. The analysis was then continued on the JDT archive. On analysing the 101 JDT questions another category of questions arose. This related to comprehension of the code: For example, *“Can anyone suggest me where about in the code is a good starting point in understanding how the component works?”* We labelled such questions *“System Implementation”* questions and they were defined as questions that refer to the workings of the code base.

Table 4.2 - The provisional information types identified during pilot study (Sharif and Buckley, 2008a)

Information Type	BSF	JDT	Total
Tool/Technology : Question about the tools or technology that they use to do their task such as the IDE etc. Example: <i>"I can't loggin to the wiki .. how can I create an account ?"</i>	8	22	29
Correction Procedure / Coding Plan : Question about how to fix bug(s). Example : <i>"Any tips on how best to do this?.. (Given an example of steps taken)"</i>	5	22	27
Legality / Protocol :Asking permission or question about team protocol in doing task. Example:” <i>There's some info on the Sun site - basically fill in a form and email it to Sun. Can I do this on behalf of Apache?"</i>	9	7	16
System Design :Questions about software design issues, user requirement and functionalities. Example: <i>"Is the implementation of generics expected to trigger major changes to the internal data structures?"</i> ; <i>"Is there any plan to add pattern functionality to the JDT UI, i.e.."</i>	1	12	13
Documentation : Questions that ask information about documentation for reference. Example: <i>"Is the syntax for these things documented anywhere?"</i>	3	7	10
Request for Help/ Bug Fix Support : Question that asking support for bug fixing or another task. Example : <i>"Any chance you'd like to submit a patch to fix it?"</i>	6	4	10
File Configuration : Question about file configuration. Example: <i>"What is the distribution directory in the src zip / tgz ?"</i>	2	6	8
Awareness : Question to make sure that the asker or the audience has up to date information about current issues in the team. Example : <i>"Do we need to tell anyone in Apache we're doing this?"</i>	5	2	7
Validation for Proceeding With Changes : Questions to ensure that changes undertaken are appropriate. Example: <i>"I see there's a JIRA issue now, and my changes would've been needed anyway, so I hope you're ok?"</i>	3	4	7
Validation for Correctness of Changes / Sufficiency of Change : Question to validate the correctness or sufficiency of current changes. Example : <i>The changes that I have made are very small (possibly a bit naive-but probably ok).What do you think?"</i>	2	2	4
Bucket : Question that doesn't fit with any above categories.		14	14

Table 4.3 - Refined of initial information type (Sharif and Buckley, 2008a)

Level I	Level II
<i>Validation</i> Asking the community to validate something.	Correctness of Changes : Questions that are asking others to verify the correctness of changes the programmer made.
	Sufficiency of Changes : Questions to ensure that the changes they are doing are sufficient or are still required to solve certain problems
<i>Procedure</i> Asking the correct procedure when doing task.	Tool / Technology : This category of question is asking how (new) tools / software are used
	Legality / Protocol : This category is asking about the protocol for doing something. Open source programmers need to follow certain protocols to, for example, make sure a programmer is qualified to contribute (Rigby and Storey, 2011).
	System : Asking for coding tips given a specific protocol
<i>Awareness</i> Asking questions about the wider teams activities	System : Sometime programmers want to update / enhance their knowledge on the state of the system with respect to what others do.
	Person / Owner : Questions about team member who is performing a certain role or asking who is the owner of a certain artifact (artifact such as a source code file or specific document)
	Support Required : This category of question is to ask another programmer to take the responsibility for a task.
<i>Technical Information:</i> Questions related to their current programming task	System Implementation : These are questions that refer to the code base
	System Documentation : Open source programmers often need to confirm something using documents such as official guidelines and / or emails.
	System Design : Questions to check the design of the software.
	File Configuration : Question about the system's configuration.
<i>Bucket Category</i> Questions that do not fit with any of above category.	

We also found one question that asked about the person who was responsible for an artifact: “*Can someone please point me to the information development team that wrote the used documentation?*” and we named this category as “*Person*”. These newly emergent categories are shown in Table 4.3 (typed in bold).

After reflection, several common features shared by two or more categories were identified. So the categories were divided into two levels, one general (level I) and one specific (level II). This is shown in Table 4.3 and can be considered a form of axial coding.

Table 4.3 shows 4 general subjects of information sought by programmers which are “*Validation*”, “*Procedure*”, “*Awareness*” and “*Technical Information*”. Each of them consists of several specific information types. For example, programmers tend to seek *Validation* on the *Correctness* and *Sufficiency* of changes that they have made. Likewise, programmers also looking for the right *Procedure* in using *Tools/Technology* to do their task and they also need to know the right procedure with respect to their team rules (*legality / protocol*).

On further reviewing this data it became apparent that 2 properties could be associated with each of the questions: the target of the request (*Information Focus*), and the type of information sought on that target (*Information Aspect*). This was initially prompted by having ‘*System*’ type information residing under 3 different “Level 1” categories where system is the target of the request, but different attributes (or aspects) of the system were being sought. For example, in the case that programmers are looking for specific lines of code that implement a certain functionality, the *Information Focus* is *System Implementation* (as the question is directed at the codebase) and the *Information Aspect* is location (locating that specific line of code). In the example mentioned here, the question aspect would be a “*where*” question. The presence of this type of question is in line with Letovsky’s (1987) study of programmers’ cognitive process during program comprehension. He refers to the distinction in question aspect as

'strategy'. Another advantage with this re-arrangement is that it allows unforeseen combinations of focus and aspect. For example, a programmer could ask for validation of parts of the system rather than for his/her change to the system. Hence, the focus/aspect distinction was adopted.

In addition, on further reflection, it became clear that programmers stated questions in very different ways, based on whether they had an existing hypothesis as to the answer or not (possibly indicating *Knowledge Strength*). A hypothesis based question might reflect a Top-Down comprehension strategy whereas a question without a hypothesis might be more indicative of Bottom-up comprehension (Brooks (1983). Hence the schema was refined, based on these 3 main dimensions;

- 1) Information Focus
- 2) Information Aspect.
- 3) Knowledge Strength.

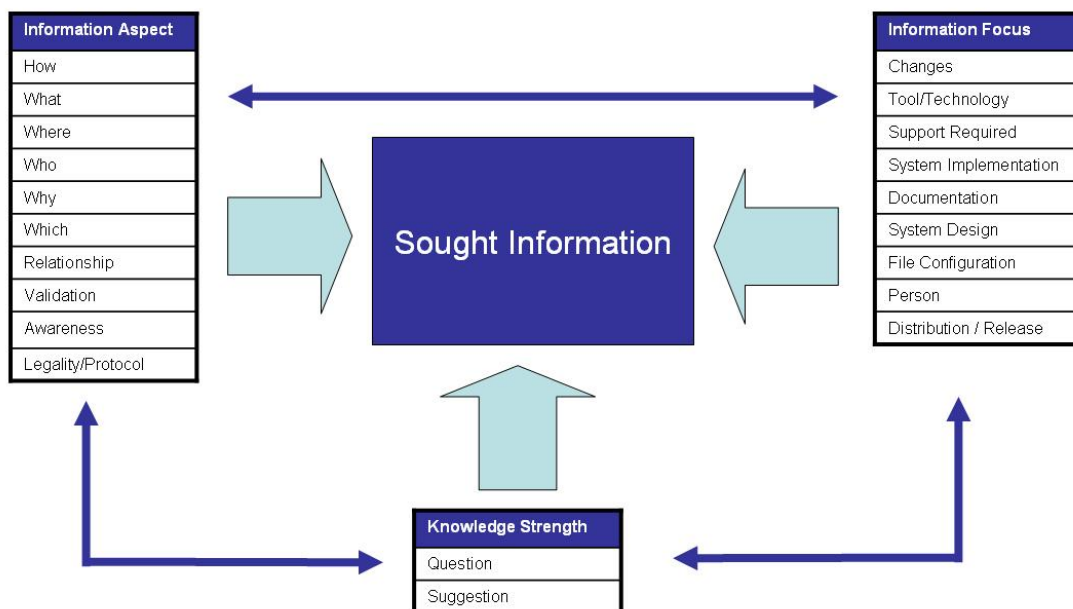


Figure 4.1 - Schema for open source programmer's information seeking

4.3.3 The Information Seeking Schema

The resultant schema from this pilot study is presented in Figure 4.1. Every question identified in the mailing list has one attribute from each of these dimensions.

4.3.4 Mapping the provisional information type into the new schema

As an illustration of the transition from the provisional information types (the initial schema) into the new schema, Table 4.4 below maps the categories of each, based on the initial dataset, showing that the new schema is a superset of the previous one.

Table 4.4 - The coverage provided by the Information Seeking Schema

Provisional Info. Type	Information Focus	Information Aspects
Validation: Correctness of changes	Change	Validation
Validation: Sufficiency of changes	Change	Validation
Procedure: Tools and Technology	Tools and Technology	How, Why, What, Where, Protocol / Legality, Which
Procedure: Protocol	Person	Protocol / Legality
Procedure: System	System Implementation	How
Awareness: System	System Implementation	Awareness
Awareness: Owner	Person	Awareness, Who
Awareness: Support required	Person	Awareness
Technology: System Implementation	System Implementation	What, How, Why and Where
Technology: Documentation	Documentation	Where
Technology: Design	System Design	Why, Where, Relationship
Technology: Configuration	Configuration	What, Relationship

Table 4.4 suggests that all categories in the provisional information types can be mapped onto the newly created schema. We revisited the entire question set in the JDT and BSF dataset and we found questions from the datasets for each of

the resultant categories, including several that could be better modelled by combining focus and aspect in novel groupings.

Table 4.4 also suggests that individual information foci can be expanded into several aspects. For example, “*System Implementation*” is expanded into 4 *Information Aspect* as shown below⁹:

- *Why* (e.g. “*I am getting an exception being thrown when trying to create new java class and I was wondering if anyone could shed any light on why?*”)
- *What* (e.g. “*Could anyone comment on what this compiler warning and its performance implication is all about? (Given a compiler warning dialog)*”)
- *How* (e.g. “*(Describes error)... Does anyone know how I can fix this?*”)
- *Where* (e.g. “*Can anyone suggest me where about in the code is a good starting point in understanding the how component works?*”)

Three of these *Information Aspects* are in line with Letovsky (1986) when he used a *think-aloud* protocol to identify ‘*how*’, ‘*why*’ and ‘*what*’ questions used by programmers when understanding code. Meanwhile, the ‘*where*’ question is in line with O’Shea (2007) suggestion that programmers look for *location* information type especially when they are working on larger code-bases.

Likewise, *Tool/Technology* also expanded to various *information aspects* such as *Why*, *What*, *How* and *Where*. There are also other aspects that mapped onto *Tool/Technology: Protocol / Legality* and *Which*. These seem to reflect the team-oriented nature of OS programmers where they are required to confirm, the usage of various technologies.

Note that Table 4.4 shows only the foci and aspect combinations that were identified in the dataset. The resultant schema is presented in the following sections.

⁹ These examples were taken from the actual dataset.

4.3.5 Information Focus

Information Focus refers to the external representation that the information search seeks information on. There were nine individual foci identified in the BSF and JDT mailing list during this observation-cycle. Table 4.5 below defines each of these and provides examples for each, taken from the dataset captured. Please note that while these seem to bear similarity to the ‘information source’ research carried out by Singer (1998), Seaman (2002) and Sousa et.al (1998), they differ, as the focus in this research is *the artifact the programmer is looking for information about*, not *the source through which they choose to acquire the information*. In this research the source through which they choose to acquire the information is always the mailing list.

Table 4.5 - Information Focus (Sharif and Buckley, 2008a)

Information Focus	Definition and Example
<i>Changes</i>	Questions that refer to changes that programmer has made or will make. . Example: <i>“Here is a patch for the changes I had to do.... Please look into it, I may have broken many exception handling policies here”.</i>
<i>Tools / Technology</i>	Questions that refer to technology or tools that the programmers use. Example: <i>“Can we use JIRA for bug reporting for this issue instead...”</i>
<i>Support Required</i>	Questions that ask another community member to take on responsibility or tasks. Example: <i>“There are 2 non-filed open issues..... Are there any taker? ”</i>
<i>System Implementation</i>	Questions that aim to understand the code. Example: <i>“(Given a situation..)I have no idea why this is happening. Please help me solve this problem“</i>
<i>Documentation</i>	Questions referring to the documentation: Example: <i>“Is there any Apache official guidelines on this?”</i>
<i>System Design</i>	Question referring to the system’s design and requirement. Example : <i>“Is the implementation of generics expected to trigger major changes to the internal data structures?” ; “Is jdt.core.jdom built on top of jdt.core.dom? Can you get to the underlying jdom model?” ; “Is there any plan to add pattern functionality to the JDT UI”</i>
<i>File Configuration</i>	Question about configuration management. Example : <i>“ What is the distribution directory in the src zip/tgz? ”</i>
<i>Person</i>	Question about the person in-charge or responsible for some task. Example : <i>“Can someone please point me to the information development team that wrote the used documentation?”</i>
<i>Distribution / Release</i>	Question about distribution or release of source code / software product to public. Example : <i>“What do people think about cutting a release of BSF V3? Having a real release instead of just a SNAPSHOT build makes it much easier for other projects to start using BSF and getting users is the best way to get BSF tested and improved.”</i>

4.3.6 Information Aspect

Information Aspect refers specifically to the type of information sought by the programmer, from the *information focus* specified. Ten *Information Aspects* were derived by open coding of the BSF and JDT OS programmers' email communication as presented in Table 4.6 below:

Table 4.6 - Information Aspect (Sharif and Buckley, 2008a)

Information Aspect	Definition and Example
<i>How</i>	<p>Questions which attempted to identify 'how some goal of the system is achieved', 'how some software tool feature is employed' and also 'how to proceed'. This could be differentiated into several sub-categories such as :</p> <ul style="list-style-type: none"> • <i>How – System (how some goal or state of the system is achieved (code based) : "How X can be 5.</i> • <i>How – Software Tool(how some software tool feature is employed) : "How do I compile my Java source in Eclipse?"</i> • <i>How – Procedure (how to proceed): "I am a little unsure how best to make a patch, I would imagine that easiest way to this accepted would be to make patch against the Head of the org.eclipse.jdt.junit at eclipse.org right?"</i>
<i>What</i>	<p>Questions which ask to define / identify something such as what source code or software tools or their features do:</p> <ul style="list-style-type: none"> • <i>What – System : "What is the value of X at this line".</i> • <i>What – Software Tool: "What is the features of GTK that can be used with Motif?"</i> • <i>What – Configuration: "What is the .rep file?"</i>
<i>Where</i>	<p>Asking about the location of software artifacts, tool etc. Example: "Where I can find the sources for plug in so I can</p>

	<i>create a patch?"</i>
<i>Who</i>	Asking for the relevant persons to seek for information or to perform a task. Example: <i>"Can someone please point me to the information development team that wrote the used documentation?"</i>
<i>Why</i>	Asking for the purpose / explanation of a system behaviour or rationale of design. <ul style="list-style-type: none"> • Why – System: <i>"I am getting an exception being thrown when trying to create new java class and I was wondering if anyone could shed any light on why?"</i> • Why – Design: <i>"Why it is called IPackageFragment and not IPackage? getPackageFragment()?"</i>
<i>Which</i>	Question that reflects a choice between one and another subject (information focus). Example <i>"can we use JIRA for bug reporting for this project instead .. Thoughts ?"</i>
<i>Relationship</i>	Relationship between 2 or more things. It differs from other questions in that it directs itself at relationships between entities rather than at entities themselves, thus suggesting 'analysis level' information (Kelly and Buckley, 2009). Example: <i>"What is the dependence between PackageFragementRoot and PackageFragment?"</i>
<i>Validation</i>	Asking permission to do something. This strategy is normally related with Legality / Protocol. It seeks permission to do something. Example: <i>"I was just looking to submit a JIRA patch for the existing BSF 3.0 prototype code but there doesn't seem to be a JIRA project for BSF. How about I set up BSF in the ASF JIRA system and we can use that for any BSF 3.0 work?"</i>
<i>Awareness</i>	Question to make sure that the asker or the audience has up to date information about current issues in the team. Example : <i>"Do we need to tell anyone in Apache we're doing this?"</i>
<i>Legality / Protocol</i>	Questions about the protocol to follow within the project. Example : <i>"Did you got the approval to contribute your work to BSF? "</i>

As shown in the Table 4.6, categories of *Information Aspect* could be differentiated into several subcategories depending on the subject of discussion, thus validating the decision to reconfigure the coding schema: the subject of discussion can now be formulated as combination of categories from *Information Aspect* and *Information Focus*.

4.3.7 Knowledge Strength

Knowledge Strength refers to level of knowledge about the information in the programmers' query. There are 2 type of *Knowledge Strength*. These are presented in Table 4.7 below:

Table 4.7- Knowledge Strength (Sharif and Buckley, 2008a)

Knowledge Strength	Definition and Example
<i>Question</i>	This type of question is a question that is asked without a proposed answer in mind. There is no evidence in this case that the person who asks this type of question knows anything about the information that he/she asked for. Example: <i>"What is the .rep file?"</i>
<i>Suggestion</i>	Suggestion is a hypothesis-based question: Questions that are asked with an idea as to their answer already in mind. That is, the question comes with a suggestion or conjecture (Mayrhauser and Vans, 1993) for the answer. This type of question is asked to validate, to confirm or to correct the provisional answer. Example: <i>"I see there's a JIRA issue now, and my changes would've been needed anyway, so I hope you're ok?"</i>

4.3.8 Schema application result and further refinement.

Tables 4.8, 4.9 and 4.10 below summarize the findings when the resultant schema was applied to the BSF and JDT project mailing list.

Table 4.8 - Information Focus of BSF and JDT

Info Focus	BSF	JDT	Total
Changes	0	3	3
Tools / Technology	10	31	41
Support Required	5	3	8
System Implementation	11	37	48
Documentation	8	7	15
System Design	1	12	13
File configuration	2	6	8
Person	0	1	1
Distribution / Release	7	1	8
Total	44	101	145

Table 4.9 - Information Aspect BSF and JDT

Info Aspect	BSF	JDT	Total
How	15	47	62
What	6	18	24
Where	1	13	14
Who	5	4	9
Why	2	9	11
Which	1	1	2
Relationship	1	1	2
Validation	2	3	5
Awareness	3	0	3
Legality / Protocol	8	5	13
Total	44	101	145

Table 4.10 - Knowledge Strength of BSF and JDT

Knowledge Strength	BSF	JDT	Total
Question	30	78	109
Suggestion	14	23	36
Total	44	101	145

Basically, the figures in Tables 4.8, 4.9 and 4.10 give information on categories' prevalence that can be used for further refinements. Table 4.8 shows that the most sought *information focus* were *System Implementation* and *Tools/Technology* with 48 and 41 requests respectively. The lowest requested information foci were *Person* (1 request) and *Changes* (3 request). In Table 4.9, the most sought *information aspects* were *How* (62) and *What* (24) categories, while the least requested was *Relationships* (2 requests) and *Which* (2 requests). Finally, Table 4.10 shows that 30% of the requests in the dataset were *suggestion* question and the other 70% is plain questions.

In the *Information Focus* dimension, the high number of requests for *System Implementation* and *Tools/Technology* information suggests that these two categories potentially contain several sub-themes. Likewise for the *How* and *What* categories in the *Information Aspect* dimension. But the high number of requests in both could possibly be subdivided in terms of the other dimensions. In this case, *How* and *What* questions could be differentiated when combined with categories from *information focus* as shown in Table 4.6 and discussed in the previous section.

In term of *Knowledge Strength*, Table 4.10 shows that 30% of the requests in the dataset were *suggestion* questions and the other 70% were plain questions. While these figures might reflect certain information seeking behaviour among OS programmers (Sharif and Buckley, 2009b), on further consideration, the *knowledge strength* is more appropriately considered as one property of the questions but not as information type per se. Therefore, it can't be considered as a dimension of information need. For example, if a programmer asked, "*Is it possible to do this with the existing SWT framework?*" the *Information Focus* is

System Design and the *Information Aspect* is *How* as he / she is actually asking “*How to do this?*” The *Knowledge Strength* may show that the programmer has a higher confidence and thus has possibly some knowledge about the subject asked in the question. But this may just be an artifact of the programmers' personalities and so, the dimension was eliminated from the schema.

4.4 Observation on Different Stages of Software Evolution (2nd Observation-cycle)

The first observation-cycle of schema development faced two external validity threats. That is, both the OS projects used for this pilot study were studied over a short time period of evolution and both were in a vertical domain that does not represent all types of OS application domain / functional categories as suggested by Feller and Fitzgerald (2002). BSF is categorized under the *System Environment* or *Development* category while JDT falls under *Development* category. These issues can be seen as limiting the coverage of this study. The final observation-cycle will attempt to cover OS application domain categories such as *Application*, *Amusement* and *User Interface*. This observation-cycle addresses the temporal issue.

It buttresses the schema by extending the timeframe of the analysis for each mailing list. The mailing list archive that served as a dataset in the first cycle were taken from JDT's second year archive and BSF's archive for its 7th year of the project, up to September 2007. In this study, the dataset for the BSF was extended to October to December 2007. The JDT dataset were taken from the archive for the first and third year. Overall, this will lead to a schema based on the first three years of the JDT initiative and the 7th year of the BSF initiative, representing early, intermediate and latter stages of OS projects. Observations from this analysis have been reported in (Sharif and Buckley, 2008b).

4.4.1 The Dataset

Table 4.11 presents the datasets used in this cycle. The JDT programmers' mailing list (JDT, 2003) was captured for 2 periods of 1 year, for 2002 (January to December) and 2004 (January to December). In addition, a continuation of the previous BSF dataset that captured email from January to September 2007 was utilized. In this cycle, the BSF programmers' mailing list (BSF, 2007) was captured from October to December 2007. The resultant dataset consisted of 328 emails, and from this dataset 147 questions asked by the programmers were manually extracted. Open coding was then applied to this dataset to indicate possible refinements to the schema.

Table 4.11 - Dataset for Second Observation-cycle

Dataset	Archive Taken	Number Of Emails	Extracted Questions
JDT 2002	January to December 2002	81	42
JDT 2004	January to December 2004	100	69
BSF 2007	October – December 2007	147	36
Total	27 Months	328	147

4.4.2 Data Analysis and Schema Refinement.

Initial Open Coding

Open coding, augmented by review sessions with the research supervisor, suggested new categories that reflected five questions in the dataset. With respect to the *Information Focus* dimension, one question asked about current stage or progress of certain activities or projects. For example: “*Are any projects failing to build?*” While the aspect for this question would be *What* (as in “What is the progress of this project?”), the focus of this question seemed similar with the *Distribution/Release* category that was concerned with progress of a task but *Distribution/Release* was specifically targeted at the software releasing plan. On closer reflection, this question is genuinely asking about progress or the current

stage of the related activity with no concern for a releasing plan. Hence, a potential new category called *Stage* might be necessary to cater this kind of questions. However, due to small number of occurrence in the dataset, it was considered not yet necessary to create a new category for the question. This was noted as a potential new category and assessed during the next observation cycle. For the time being, due to the similarity mentioned earlier, this question was categorized under *Distribution/Release*.

Regarding the *Information Aspect* dimension, two categories of questions initially emerged from the data

- i) Question that instructed the audience to take action on the *Information Focus*. Example: “could you please be so kind and incorporate that fix into the current beta of bsf3 as well?”
- ii) Question asking about timeline or time of occurrence. Example: *When is the next BSF release expected?*

However on closer examination of the first category, these “instruction” type questions could be regarded as being of ‘*request*’ *Aspect*. They could also be considered a sub-type of ‘*Support/Required*’ type *Focus* and of *Aspect Who* questions (as in “*Who will do the change?*”). Hence, it was considered not yet necessary to commit to a new category for these questions. The second new category was named as a ‘*When*’ aspect since the evidence found in the dataset clearly asks about time-related information. Table 4.12 below presents a description for the newly suggested category.

Table 4.12 - New Categories of questions in bucket.

Dimension	Suggested Category	Definition and Example
<i>Information Aspect</i>	<i>When</i>	Questions asking about timeline or time of occurrence. Example : <i>When is the next BSF release expected?</i>

Schema Assessment

After the initial round of open coding the existing schema was re-applied to assess the degree to which it accommodated all the questions extracted from the new dataset. As a result prevalence information was determined for each of the existing categories was assessed. The results of this analysis are presented in Table 4.13 and Table 4.14.

**Table 4.13 - Content Analysis Result for Information Focus Dimension
(2nd Observation-cycle).**

Info Focus	JDT 2002	JDT 2004	BSF 2007	Total
System Implementation	8	29	10	47
Tools / Technology	19	20	6	45
Documentation	6	8	5	19
Support Required	4	0	6	10
System Design	2	7	0	9
Changes	0	2	3	5
File Configuration	3	1	1	5
Person	0	1	1	2
Distribution / Release	0	1	4	5
Total	42	69	36	147

Overall, the results shown in Table 4.13 are similar to the results of the first observation-cycle: few instances of the *Person* category and frequent occurrences for *System Implementation* and *Tools/Technology* categories. Likewise, Table 4.14 showing few instances for the *Relationship* category and higher information requests for *How* and *What* categories.

Table 4.14 - Content Analysis Result for Information Aspect Dimension (2nd Observation-cycle).

Info Aspect	JDT 2002	JDT 2004	BSF 2007	Total
How	18	24	5	47
What	9	17	9	35
Where	5	11	3	19
Why	3	10	4	17
Who	3	2	7	12
Validation	1	3	1	5
Awareness	1	1	3	5
Which	2	0	0	2
Legality / Protocol	0	0	2	2
When	0	0	2	2
Relationship	0	1	0	1
Total	42	69	36	147

High Requested Questions

As mentioned in previous section, the high number of requests obtained by *System Implementation* and *Tools/Technology* suggests that these two categories potentially contain prevalent sub-themes. This is in line with the Grounded Theory analysis phase called axial coding (Charmaz, 2006) where sub-categories can be identified.

Deeper analysis of these categories revealed that *System implementation* can refer to both code comprehensions related questions and task related questions. Hence, we created more detailed categories to replace the *System Implementation* category. These are *System Enhancement*, *System Bug*, *Task-Implementation* and *Task-Test*. Indeed, in doing so, the ‘Change’ questions migrated to these categories. The description of these newly suggested categories is presented in Table 4.15 below:

Table 4.15 - Finer Grained Categories of System Implementation category.

Information Focus	Definition and Example
<i>System Enhancement</i>	Questions that aim to understand the code in order to make evolutionary change. Example : <i>"...but I need to understand the refactoring currently in Eclipse now. Can anyone suggest me where about in the code is a good starting point in understanding how the component works "</i>
<i>System Bug</i>	Questions that aim to understand the code in order to trace a bug. Example : <i>"(Given an error..)I have no idea why this is happening. Please help me solve this problem"</i>
<i>Task-Test</i>	Question related to testing. Example : <i>"Is renameParticipants working? I am doing a spike test but cannot get it work so far."</i>
<i>Task-Implementation</i>	Question about tasks that are related to Implementation. Note that this is not about comprehending the code but more directed at the task to be undertaken. Example : <i>"Maybe you need to post more code, or maybe you need to update ecs-1.4.1?"</i>

Reanalysis was also done to *Tools/Technology* questions to see any possible sub-themes. On closer review of all the questions categorized under this category, we found that the questions asked about three different types of tools which are *Integrated Development Environments (IDE)*, *Operating Environments*, and *Communication Channel*. Hence we created three new categories as finer grained versions of the *Tools/Technology* category (as described in Table 4.16 below):

Table 4.16 - Finer Grained Categories of Tools/Technology category.

Information Focus	Definition and Example
<i>Integrated Development Environment (IDE)</i>	<p>Question asking about the IDE used in the project. These could be differentiated into 2 main aspects when combined with <i>Information Aspect</i> categories :</p> <p>IDE – Guide. Question asking for detailed guidance on <i>How</i> to use the IDE component to achieve a goal. For example: “<i>How to use Eclipse (for java) with CVS.</i>”</p> <p>IDE - Feature. Question that ask about features of the IDE component. Normally these are asked as <i>What</i> questions such as: “<i>I was wondering if Eclipse, has the capability for enabling the development of the EJBs and deploying it in the container for the any application sever that might come with it.</i>” This would also be asked as <i>Where</i> questions such as “<i>where can i configure quickDiff in eclipse 3.0m7?</i>”</p>
<i>Communication Channel</i>	<p>Question that refer to the communication channel such as mailing lists, or bug reporting channels. These could again be differentiated into 2 main aspects:</p> <p><i>Communication Channel</i> – Guide. Question that ask for detailed guidance on <i>How</i> to use the channel. For example : <i>Is there something that needs to be done to get the SVN commits i do have notifications sent to the BSF dev list?</i></p> <p><i>Communication Channel-Choice</i>. Questions reflecting a choice between one or another channel. These are normally asked as <i>Which</i> questions. For example: “<i>can we use JIRA for bug reporting for this project instead .. Thoughts ?</i>”</p>
<i>Operation Environment</i>	<p>Question asking about the operation environment of the IDE, server and any related surrounding technical context that is involved in the application running such as its plug-ins. Likewise, this can be differentiated into guidance and choice:</p> <p><i>Operation Environment</i> – Guide. Question that ask for detailed guidance on using specific components for successful execution of the application. For example: “<i>Can I invoke update manager using bash scripts to install our plugins and features?..</i>”</p> <p><i>Operation Environment</i> - Choice. Question that reflects a choice between one or another operation environment. Normally asked in <i>Which</i> questions. For example : “<i>Is Apache easier to deal with than Jakarta?</i>”</p>

There are also prevalent question types in the *Information Aspect* dimension, such as *How* (47 request) and *What* (35 request) questions. However, for frequent *What* and *How* questions, it was not deemed necessary to divide them into finer grained categories, as these categories were already well distinguished by their *Information Focus*.

Low Requested Questions

Table 4.13 and 4.14 also shows categories with a very low number of requests. In the *Information Focus* dimension, the lowest requested question category was *Person* (2 request). In table 4.14, the least requested categories were *Which* (2 request), *Legality / Protocol* (2 request), *When* (2 request) and *Relationship* category (1 request).

The low request of *Which* and *Legality / Protocol* type questions in this dataset doesn't suggest elimination as they were strongly prevalent in the previous analysis. However, the *Person* and *Relationship* category seems to have maintained a low request rate through the 2 observation-cycle. Hence these two categories were placed on a watch-list for potential elimination.

For *Person* focus, it frequently has a 1:1 relationship with the *Who* aspect, suggesting a certain amount of redundancy. This overlap with *Who* questions is potentially interesting. For example, it is strange when the *Person* category got a low number of requests (3) that *Who* questions were more frequently asked (6 request). Closer investigation of *Who* questions can give some insight on this issue. *Who* questions not only referred to persons (as in “*Can someone please point me to the information development team that wrote the used documentation?*”), but could also refer to another *Information Focus* such as *Tools/Technology* (example, “*Is anyone debugging Java with Eclipse?*”) and *Support Required* (example, “*...in<http://....> a request was made to change the jython engine in BSF 2.4 such that the apply() method would forward the arguments. Would there be any takers on doing the necessary work on this?*”). Hence the higher number of request obtained by the *Who* category is because of

the coverage of this category is not limited to *Person* aspect questions only but it also covers various other *information foci*.

There is also a validity issue for this round of study that might explain the low occurrences noted for certain categories. This study looks at questions posted on two open source programmers' mailing lists consisting of 328 emails containing 147 questions. Even though this is a large number, the data is analyzed into 10 and 11 categories in 2 different dimensions respectively. Thus, the number of data points in each category varies from 0 to 29, with 87% of the 10 categories in *Information Focus* (Table 4.13) having less than 10 questions in total. This can be seen as a validity threat of this observation-cycle. Likewise, as shown in Table 4.13 and Table 4.14, approximately 40% of the *Information Focus* and *Aspect* categories across the entire projects got 0 or 1 request. On closer review, most of the 0 and 1 request columns represent the early (JDT 2002 column) and intermediate (JDT 2004 column) evolution stages of the JDT dataset, suggesting an influence of software development stage in programmers' information seeking behaviour. As a consequence, this finding, of limited *person* and *relationship* questions, is based on quite weak evidence. Further investigations, with larger datasets, will provide more definitive evidence for the inclusion of each category and such an analysis is presented on the next refinement phase. Consequently, these two categories were retained in the schema for the next observation-cycle on a much larger dataset. The schema, as of the end of this observation-cycle is contained in Appendix A. The newly created categories are written in bold.

4.5 Stress Testing the Schema with a Large Dataset (3rd Observation-cycle)

The rationale for this observation-cycle is to provide a larger dataset with which to buttress the 2nd observation-cycle, as some of the information foci seemed peripheral in that study but, due to a limited number of questions in the dataset, there wasn't sufficient evidence for their omission.

4.5.1 Method and the Dataset

This observation-cycle enlarges the dataset taken from the BSF and considers a new OS project called the ECS project. The ECS or Element Construction Set (ECS) is an OS project to create a Java API for generating elements for various markup languages. The ECS will allow a user to use Java Objects to generate markup code. The project was chosen as it has long period (eight years) of email archive and so has higher coverage of various development stages. The dataset used in this study is shown in Table 4.17.

Table 4.17 – Dataset for 3rd Observation-cycle

Dataset	Archive Taken	Number Of Emails	Extracted Questions
BSF 2003-2004	Jan. 2003 - Dec. 2004	391	102
ECS 2001-2008	Jan. 2001- Dec. 2008	398	80
Total	120 months	789	182

The BSF developers' mailing was captured from an early stage of the project (the first 2 years of that archive). The ECS mailing list was captured for 8 years from 2001 to 2008 (from the beginning of that archive to the last archive at the time of analysis. These periods were chosen to provide more encompassing time-frames and a larger dataset for stress-testing the schema (Sharif and Buckley, 2009b, Sharif and Buckley, 2009a)

The resultant dataset consisted of 789 emails, and from this dataset 182 questions asked by the programmers were manually extracted. In this observation-cycle, content analysis is applied to the dataset according to the schema detailed in Appendix A (from observation-cycle 2). Bucket category questions were then inspected to identify newly emergent categories and prevalence information for each category was then used to refine the schema.

4.5.2 Results and Schema Refinement.

The overall prevalence results of the study are presented in Tables 4.18 and Table 4.19. The BSF columns in each dimension reports on emails archive for 2003 and 2004. Likewise, the ECS column reports on emails archive from 2001 to 2008.

**Table 4.18 - Content Analysis Result for Information Focus Dimension
(3rd Observation-cycle)**

Info Focus	BSF	ECS	Total
Task-Implementation	17	15	32
System Enhancement	15	11	26
Support Required	13	8	21
System Design	16	2	18
System Bug	10	7	17
IDE	8	8	16
Documentation	3	12	15
Distribution / Release	5	5	10
File Configuration	4	4	8
Operation Environment	4	1	5
Task-Test	3	2	5
Communication Channel	0	1	1
Person	0	0	0
<i>Bucket</i>	4	4	8

**Table 4.19 - Content Analysis Result for Information Aspect Dimension
(3rd Observation-cycle).**

Information Aspect	BSF	ECS	Total
What	32	14	46
How	27	12	39
Awareness	7	20	27
Why	9	13	22
Who	13	8	21
Where	6	4	10
Validation	3	7	10
When	4	0	4
Which	1	1	2
Legality / Protocol	0	1	1
Relationship	0	0	0
Bucket	0	0	0

Overall the tables present supporting evidence for the inclusion of many of the categories in OS programmers information seeking.

High Requested Questions

The data shown in Table 4.18 represents a more even distribution, as the highly requested categories reported in previous observation cycle (*System Implementation* and *Tools/Technology*) are now divided into several finer grained categories. The most sought *information foci* as shown in Table 4.18 were *Task Implementation* and *System Enhancement* with requests numbering 32 and 26 respectively. Closer examination on the related questions suggested no other sub-themes.

Low Requested Questions

There are a few question types with a very low frequency. In Table 4.18, there were no requests for *Person* and there was only 1 request for the *Legality / Protocol* category shown in Table 4.19. The low request rate for *Legality / Protocol* in this dataset is only considered a 'warning flag' in this observation-

cycle as it has been requested frequently in another dataset (especially during the first observation-cycle). However, this will be noted and re-assessed in the final cycle.

In the other hand, the low request rate for the *Person* category seems to maintain the same trend as the other datasets and this category is withdrawn from the schema. While it is noted that this category was in evidence (Seaman 2002) and may inform on programmers information seeking, the very low request for this category in all three datasets implies it is of little relevance (in general) to open source programmers.

Likewise, in the *Information Aspect* dimension, there was no request for the *Relationship* aspects. Again, it should be noted that this category would seem to be relevant for programmers but it is withdrawn due to the infrequent nature of this type of question over the datasets, suggesting little relevance to OS programmers.

Bucket Analysis

Eight questions that didn't fit with any category in the current schema were found in the dataset. The questions suggested two new types of question as discussed below:

- i) Seven questions were referring to stage of certain activities or project. (E.g. *"I've not seen any noise on this list since I joined. Is there any life in the BSF sub-project?"*). While the question aspect for this question is *What* (as in "What is the stage of this project?"), the newly emerged focus could be referred as *Stage*. This type of question was first found during the 2nd observation-cycle (see section 4.4.2) but was later categorized under existing *Distribution / Release* category due to its limited number of occurrence during the study. However, the figure of 7 requests found here, confirmed the relevance of this newly emergent category in the schema. Hence, new category named *Stage* was added into the schema.

- ii) One question referred to coding conventions: “*Is there a preferred coding standard?*” While the *Information Aspect* for this question is *What* (as in “*What is the coding standard for this..?*”), the newly emerged *Focus* category could be referred to as *Coding Standard*. Content analysis showed that only one request for this type of question occurred, suggesting a weak relevance. Review of the other datasets confirmed this impression. Hence, no new category was created for this type of question in the schema, but it was noted and will be watched out for during the next analysis.

The schema has been updated according to the results of this analysis.

4.6 Observation on Various Categories of OSS Projects (4th Observation-cycle).

The previous cycles were done without specifically addressing coverage on different open source project categories. Specifically, all of the datasets were chosen based on their evolution stage and the activeness of their email communications, as recorded in their project archive. Given the various types of open source software projects (see chapter 2, section 2.6.4), it is important to carry out representative sampling and extend this study to cover the different categories of OS projects being developed. Hence, this cycle is aimed at maximizing coverage of OS project types, by observing open source programmers mailing list from different categories of open source project.

4.6.1 The Dataset

The cycle described in this section employs Daniel et al's (2009a) open source project categorization, as a basis for the OS, mailing-list, dataset. Personal discussion with one of the authors was undertaken for this purpose, as specific examples of OS projects in each category were not always identified in the paper

(see Appendix A). Based on Daniel et al's categories, and the personal communication, this cycle analysed the questions contained in 3 mailing list: Eboard¹⁰, Resiprocate¹¹ and SwingWT¹². Table 4.20 below characterizes the mailing list used as the dataset in this cycle.

Table 4.20 - Datasets used in 4th Observation-cycle.

OS Project	Description	Period	# of Emails	# of Questions
Eboard	User-friendly chess interface for ICS (Internet Chess Servers)	1 year (2001)	182	45
SwingWT	Implementation of the Java Swing and AWT APIs	1 year (2004)	302	108
Resiprocate	Dedicated to maintaining a complete, correct, and commercially usable implementation of SIP (Session Initiation Protocol)	1 year (2009)	228	107

The dataset used in this study is a dataset that reflects the successful OS software project categories identified by Darcy (2009). There are 2 main reasons for this choice:

- Successful OS software project would have more active maintenance activity, and hence more relevant information seeking evolutionary lifecycles.
- Successful OS software projects would have longer maintenance cycles, and are thus the projects that might benefit from application of the resultant schema.

¹⁰ http://sourceforge.net/mailarchive/forum.php?thread_name=Pine.LNX.4.30.0104290400270.19983-00000%40moriamit.edu&forum_name=eboard-devel

¹¹ <http://list.resiprocate.org/archive/resiprocate-devel/mail3.html>

¹² http://sourceforge.net/mailarchive/forum.php?forum_name=swingwtdevelopers&max_rows=25&style=threaded&viewmonth=200401

EBoard is a user-friendly chess interface for ICS (Internet Chess Servers) (Sourceforge, 2001). This open source project is categorized as a *User Centered* project (Darcy, 2009, Daniel et al., 2009a). Projects in the *User Centered* category are successful in term of usage and development activities and this project is deemed successful because of its high number of downloads, high number of bug report activities and high number of total of releases (Daniel et al., 2009a). Emails taken from Eboard's mailing list for this cycle are from its year-2001 archive. It was the first year of the project and contains the most active email period of the archive.

SwingWT is an implementation of the Java Swing and AWT APIs. This allows existing Java/AWT/Swing applications to be compiled natively under Linux, or allows native widgets for existing Swing/AWT applications (Sourceforge, 2004). SwingWT is categorized as being in the *Controlled* category based on its fluctuating structural complexity, as reported in (Nasseri, 2009), its longevity and its high rates of download, as reported in (Sourceforge, 2007). This is in line with Daniel et al's (2009a) description of the *Controlled* category: successful, of high growth in size while having its structural complexity fluctuating. Emails taken from SwingWT's mailing list for this cycle are from its year 2004 archive. That year was the first year of the project and is the most active in terms of email communication.

The Resiprocate project is dedicated to maintaining a complete, correct, and commercially usable implementation of SIP (Session Initiation Protocol) and a few related protocols (Resiprocate, 2009). The Session Initiation Protocol (SIP) is a signalling protocol (Rosenberg et al., 2002), widely used for controlling multimedia communication sessions such as voice and video calls over Internet Protocol (IP) (Wikipedia, 2008). This project is categorized as being in the *Counter-cultural* category (Darcy, 2009, Daniel et al., 2009a). *Counter-cultural* projects are relatively successful in attracting development activity but not in attracting user interest (Daniel et al., 2009a). The Resiprocate project fits into this category because it has an average number of developers and continues to be active after the first year. However this project does not attract a very high

number of downloads (Daniel et al., 2009a). The archive used for this cycle is from year 2009, the latest archive during the study. Table 4.21 below summarizes the categories of these OS projects.

Table 4.21 - Category for each mailing list used in 4th Observation.

OS Project	Category	Characteristic
Eboard	<i>User Centered</i>	Successful in term of usage and development activities : <ul style="list-style-type: none"> ○ Increased size and structural complexity. ○ High download rate.
SwingWT	<i>Controlled</i>	Quite successful across usage indicators and development indicators. <ul style="list-style-type: none"> ○ High growth in size while structural complexity decreased or fluctuated. ○ High download rate.
Resiprocate	<i>Counter-cultural</i>	Successful in attracting a development community but not in attracting user interest. <ul style="list-style-type: none"> ○ Fluctuating size and increasing structural complexity. ○ Comparatively low download rate.

It should be noted that, while Feller and Fitzgerald’s (2002) categorization (see section 2.6.4) was not explicitly addressed in this work, we have broadened the schema’s applicability in terms of their categorization also, as shown in Table 4.22.

Table 4.22 – The schema applicability on Feller and Fitzgerald’s (2002) Categories

Feller and Fitzgerald’s (2002) Categories	OS Project
System Environment	BSF, ECS, SwingWT, Resiprocate
Development	JDT , ECS, SwingWT
Application	JDT
User Interface	Eboard
Amusement	Eboard

4.6.2 Results and Schema Refinement

260 questions were extracted from 712 emails in these mailing lists. The results of the cycle are presented in Table 4.23 and Table 4.24 below.

**Table 4.23 - Content Analysis Result for Information Focus Dimension
(4th Observation-cycle)**

Info. Focus	Eboard	Resiprocate	SwingWT	Total
System Bug	6	18	30	54
System Enhancement	9	30	9	48
Task-Implementation	5	5	27	37
System Design	6	16	9	31
Support Required	4	11	8	23
Communication Channel	3	10	1	14
Integrated Development Environment (IDE)	1	1	10	12
Operation Environment	4	4	3	11
Documentation	2	3	2	7
Distribution / Release	2	1	4	7
Stage	0	6	1	7
Task-Testing	2	1	2	5
File Configuration	1	1	2	4
Bucket	0	0	0	0
<i>Total</i>	<i>45</i>	<i>107</i>	<i>108</i>	<i>260</i>

**Table 4.24 - Content Analysis Result for Information Aspect Dimension
(4th Observation-cycle)**

Info. Aspect	Eboard	Resiprocate	SwingWT	Total
How	13	40	27	80
What	10	31	17	58
Why	10	9	28	47
Who	4	14	14	32
Awareness	4	6	6	16
Where	0	6	5	11
Legality / Protocol	2	0	7	9
Validation	1	0	2	3
Which	0	1	1	2
When	1	0	1	2
Bucket	0	0	0	0
<i>Total</i>	<i>45</i>	<i>107</i>	<i>108</i>	<i>260</i>

High Requested Questions

Overall, Table 4.23 shows strong emphasis on implementation based categories. *System Bug* (54 request), *System Enhancement* (48 request) and *Task Implementation* (37 request) were all frequently observed. In terms of schema refinement, closer examination has been done to see any possible sub categories for these highly requested question types. However, this closer examination suggested that there were no further sub-themes in the data and that the programmers just emphasized these concerns. This was also true of the how (80 request), what (58 request) and why (47 request) categories in Table 4.24.

Low Requested Questions

Several categories show low numbers of requests with a request rate of less than 1 % of the total requests in the dataset. The low request categories are *Which* (2 request) and *When* (2 request). To give a more holistic view of the frequency of these and other low-request categories, Table 4.25 presents their frequency across all observation-cycles.

Table 4.25 – Low requested Information Focus

Low Requested Categories	Number of Request During Observation-cycle			
	1st	2nd	3rd	4th
Person	1	2	0	0
Which	2	2	1	2
Relationship	2	1	0	0
When	0	2 (retrospective)	4	2
Coding Convention	0	0	1 (retrospective)	0

The figures shown in Table 4.25 justify the decision to exclude coding conventions, person and relationships from the schema, as taken during the third cycle. However, *Which* and *When*, although they have low numbers of requests, were more consistently shown over the cycles. Specifically, *Which* questions

were identified in all cycles while *When* questions were consistently observed since they were first identified during the 2nd observation. In the author's opinion, this justifies the retention of the two categories in the schema.

Bucket Analysis

No question was categorized under the bucket category during this cycle. Hence no new category was discovered (in either dimension) for the schema. Given the sample size that expanded over all cycles the emergent categories seem increasingly resistant to change. This suggests that the schema is achieving a degree of theoretical saturation in Grounded Theory terminology (Power, 2009). That is, the resultant schema is considered finalized (saturated). This finalized schema is presented in Figure 4.2.

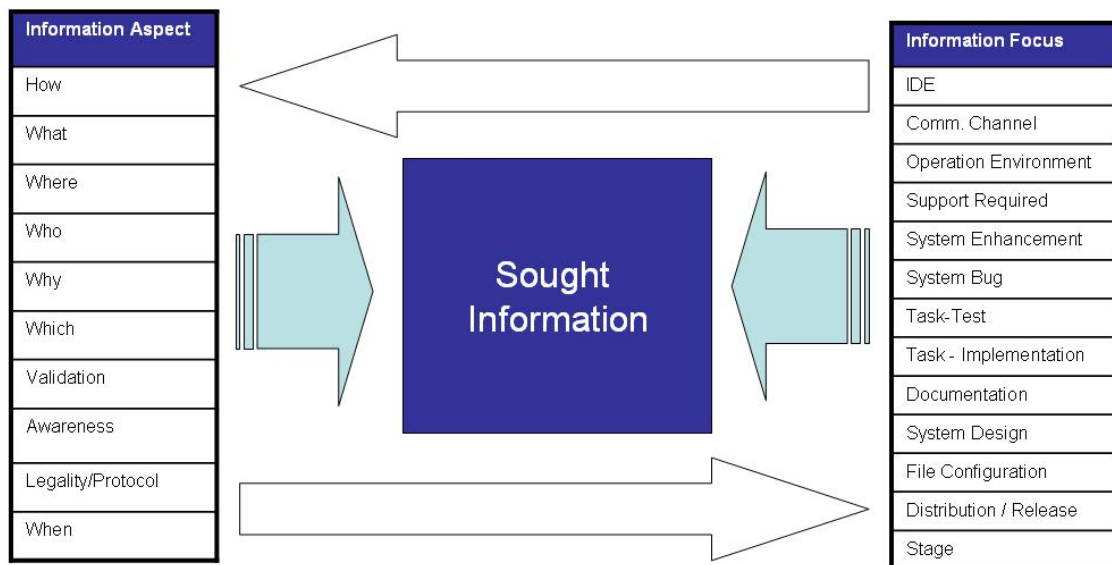


Figure 4.2 – Finalized Schema for open source programmer’s information seeking

In general, the finalized schema consists of two dimensions: namely *Information Focus* and *Information Aspect*. Every question identified in the mailing list has one attribute from each of these dimensions. The composition of these two dimensions produces a rich variety of information seeking requests across the datasets studied. Detailed definitions of each category and examples are shown in Appendix B.

4.7 Conclusion

The sampling process employed in this study has given us wide sampling with respect to stage of evolution and Darcy's categorization of OS project characteristics. It has also provided wide (if not total) coverage of Feller and Fitzgerald's categorization.

The development of the schema started with an open coding procedure done on initial dataset. Then the resultant schema was iteratively refined over different datasets using a combination of open coding and content-analysis based coding. Results of this cyclical analysis were used to reassess and refine the schema in every cycle. During the last cycle, there were no new emergent categories identified, even though the dataset increased substantially and different types of OS projects were considered.

In reassessing all categories that were infrequently observed in previous phases, some of the categories were confirmed as not of general importance for OS programmers, and were therefore eliminated. At this stage, it is expected that the resultant schema is largely resistant to change. Thus, this schema will be employed to reanalyse the entire dataset employed in the schema refinement cycles, to identify findings with respect to frequency of information needs, response rates to frequent information needs and trends over projects and time. This will be presented in the following chapter.

Chapter 5

Schema Application

CHAPTER 5 Schema Application

5.1 Introduction

This chapter will present the application of the schema to reanalyse all the datasets used in the previous phases. The goal of this analysis is to investigate OS programmers' information needs based on their frequency of request and the response rates to frequent needs. The investigation will also identify if there are trends in OS programmers information seeking across projects. Content analysis on the entire datasets is used to these ends.

5.2 Empirical Study

5.2.1 Objectives

The study described in this section is an attempt to apply the resultant schema to OS programmers' information needs. Below are the objectives of this analysis:

- To investigate OS programmers' information needs based on their frequency of request.
- To identify if there are trends in OS programmers' information seeking across projects.
- To investigate the response rate and the response frequency for different types of questions, thus informing further on their information needs.

5.2.2 Dataset.

Overall, the dataset used in this research consists of 17 (yearly) archives taken from six OSS projects. This data set resulted in 2104 email communications from which 739 questions were extracted. Table 5.1 and 5.2 below show this dataset.

Table 5.1 – Description for OSS projects used in this thesis.

OSS Project		Description
1.	BSF	The Java Bean Scripting Framework (BSF) is an OS project concerned with allowing Java applications to contain embedded languages, through an API to scripting engines.
2.	JDT	Java Development Tool (JDT) is an OS project concerned with enabling Eclipse for Java development.
3.	ECS	The Element Construction Set (ECS) is an OS project to create Java APIs for generating elements for various markup languages. The ECS will allow a user to use Java Objects to generate markup code.
4.	Eboard	A user-friendly chess interface for ICS (Internet Chess Servers)
5.	SwingWT	Implementation of the Java Swing and AWT APIs
6.	Resiprocate	Dedicated to maintaining a complete, correct, and commercially usable implementation of SIP (Session Initiation Protocol)

Table 5.2 – All dataset for this thesis.

Email Archive	Archive Taken	Number of Emails	Extracted Questions
BSF	January 2003 - December 2004 January - December 2007 (36 Months)	666	187
JDT	January 2002- December 2004 (36 Months)	328	212
ECS	January 2001 – December 2008 (96 Months)	398	80
Eboard	January – December 2001 (12 Months)	182	45
SwingWT	January – December 2004 (12 Months)	302	108
Resiprocate	January – December 2009 (12 Months)	228	107
Total	17 Years (204 Months)	2104	739

All of these datasets were used in the modelling and refinement iterations discussed in chapter 4. As mentioned previously, the OS projects and timeframes showed in Table 5.2 were chosen to maximize the coverage of this research in terms of:

- i. Observation on different stages of software evolution (see section 4.4). This dataset covers a time-span of eight years for ECS project, three years for JDT and BSF project and one year for Eboard, SwingWT and Resiprocate project.
- ii. Observation on various categories of OS project (see section 4.6). The dataset covers the three successful OS categories detailed in Daniel's (2009a) taxonomy; categories that suggest a longer maintenance phase. It also covers five out of six categories from Feller & Fitzgeralds'(2002) taxonomy, suggesting high coverage of the domain.

5.2.1 Protocol

The analysis work involved revisiting and recategorizing each of the questions in the spreadsheet generated during schema refinement. The researcher revisited all of the questions and carried out the categorizing analysis using a content analysis procedure with the aid of a coding schema. For this purpose, the resultant schema from chapter 4 served as the coding schema for the analysis.

The researcher also revisited all the original emails of the questions in the dataset (in their email archive website) to analyse responses for each of the questions. This analysis work involved tracing all emails that responded to the original emails, counting the number of responding emails and recording the time when the original emails (asking the question) and responding emails (first and the last response email) were sent. All of the information was then recorded in the spreadsheet along with the related questions.

At the end of the analysis, discussions with another researcher (the research supervisor) were undertaken to review the findings.

5.3 Analysis Result

The analysis results are presented in Table 5.3 and 5.4. These two tables show the results (ranked by total number of requests) for all the datasets used in this research. Specifically, Table 5.3 reports on *Information Focus* and Table 5.4 reports on *Information Aspect*.

Table 5.3 – Reanalysis Result for Information Focus Categories

Info Focus	BSF	JDT	ECS	Eboard	SwingWT	Resiprocate	TOTAL
System Enhancement	18	28	11	9	9	31	106
Task-Implementation	32	19	16	5	27	5	104
System Bug	13	25	7	6	30	18	99
System Design	17	21	2	6	9	16	71
IDE	10	39	8	1	10	1	69
Support Required	28	8	8	4	8	11	67
System Documentation	18	21	12	2	2	3	58
Communication Channel	17	8	1	3	1	10	40
Operation Environment	4	24	1	4	3	4	40
Distribution / Release	16	1	5	2	4	1	29
File Configuration	7	11	4	1	2	1	26
Task-Test	3	6	2	2	3	1	17
Stage	4	1	3	0	0	5	13

Table 5.4 - Reanalysis Result for Information Aspect Categories

Information Aspect	BSF	JDT	ECS	Eboard	SwingWT	Resiprocate	Total
How	45	90	12	13	27	40	227
What	44	41	13	10	17	31	156
Why	15	23	13	10	28	9	98
Who	29	7	8	4	14	14	76
Awareness	15	6	20	4	6	6	57
Where	12	28	4	0	5	6	55
Legality / Protocol	11	8	2	2	7	0	30
Validation	8	7	7	1	2	0	25
When	6	0	0	1	1	0	8
Which	2	2	1	0	1	1	7

Table 5.5 presents an analysis of the responses received for the most popular query types posted by the OS programmers on the mailing lists. The categories presented in both tables were chosen based on their high number of requests. The first column reports on the information sought (its focus and its aspect). Column two reports on the number of queries identified for each information-type and column three presents this as a percentage of the whole. Column four shows the percentage of these queries that received a response and column five reports on the average number of responses received, for queries that received at least one response. Finally column six reports on the average number of days which passed between the query being posted and the final response, again for those queries that received at least one response.

Table 5.5 – Response Analysis Result for Information Focus and Aspect Dimension

Information Focus	Total No of Question	% of Total Request	% Answered	Average No of Response	Average Timespan of Response (days)
System Enhancement	106	14.3	64.2	2.4	1.5
Task - Implementation	104	14.1	63.5	2.8	2.9
System Bug	99	13.4	55.6	2.4	3.0
System Design	71	9.61	69.0	2.6	2.8
IDE	69	9.34	62.3	2.4	1.6
Support Required	67	9.1	55.2	2.8	3.6
Documentation	58	7.8	58.6	2.6	3.4
Information Aspect					
How	227	30.7	58.6	2.3	3.3
What	156	21.1	59.6	2.7	1.6
Why	98	13.3	67.3	2.2	1.9
Who	76	10.3	60.5	2.8	3.3
Awareness	57	7.7	57.9	2.9	1.9
Where	55	7.4	60.0	2.8	4.1
Average			61.3	2.5	2.5

5.4 Result Discussion

5.4.1 Information Focus

Constant Ranking

The graph in Figure 5.1 presents the number of *Information Focus* requests in the entire datasets. The categories of *Information Focus* were ordered according to overall rank. Each line that representing specific OS project were presented in stacked fashion to analyse overall pattern of information request. The graph shows a fairly consistent pattern of *Information Focus* across all projects with only slight divergences in different system. This shows that the resultant ranking can be largely generalized across various OS project, suggesting high generality of the schema ranking in characterizing OS programmers' information seeking.

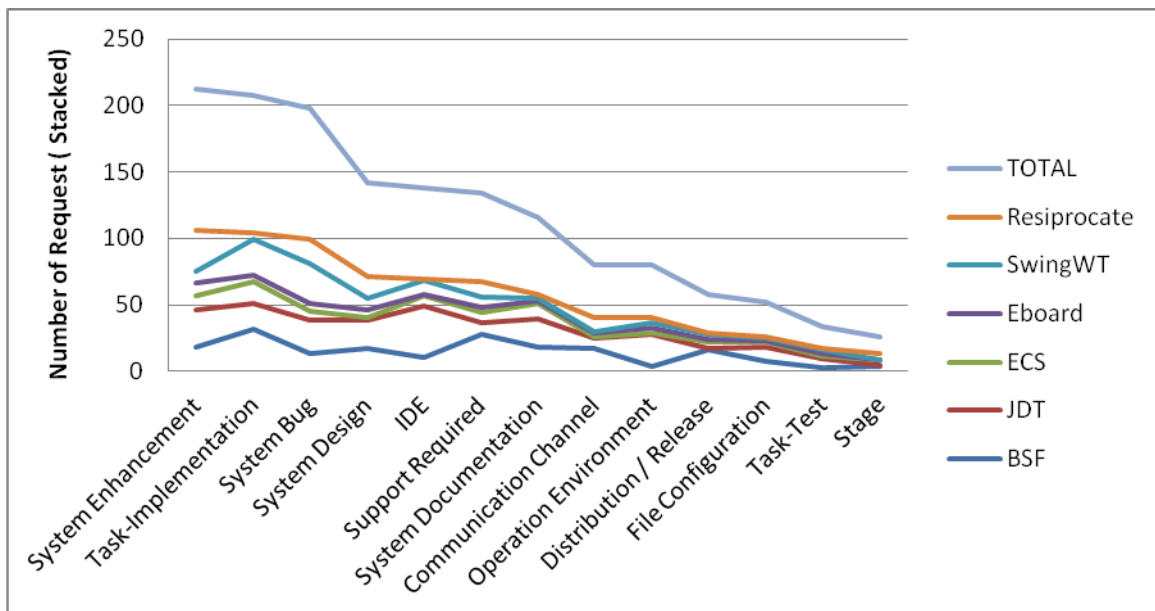


Figure 5.1 – Information Focus for all OS project in the dataset.

On closer viewing, the graphs for several of the categories don't always adhere to this ranking though. For example, the first three categories (*System Enhancement*, *Task Implementation* and *System Bug*) are slightly less consistent, as the ranks for each of these three categories fluctuate between

projects: From Figure 5.1, *System Enhancement* was often 2nd ranked to *Task Implementation* and was 3rd ranked in one case (SwingWT).

Implementation Centric

Figure 5.1 and Table 5.3 both show that OS programmers' information seeking is very implementation centric. Taking *System Enhancement*, *Task Implementation* and *System Bug* as the only categories that reflect a low-level, implementation-centric focus, these three categories were requested most frequently across all projects in the dataset. This is in line with other research (Sousa et al 1998, Singer et al 1998), which states that much of programmers' information seeking is directed at the systems' implementations. Indeed, taking *System Enhancement*, *Task Implementation* and *System Bug* as reflecting a focus on the code base, 42% of all questions were directed at the code base.

System Design

This graph also shows a strong emphasis on *System Design*, which is contra to much of the findings in the literature regarding developers' implementation focus. This finding might reflect the OS Software Development Life Cycle according to Feller & Fitzgerald (2002) who state that planning, analysis and design stages are concatenated and performed typically by a single developer or small core group. These design decisions are generally made in advance, before the larger pool of developers starts to contribute. Hence, most of OS programmers' contribution is directed at the systems' implementations but, since the design decisions were made in advance, it also possible that OS programmers need to look for information about the design. This could be the reason for the considerable proportion for *System Design* question (9.6%) in the maintenance phases of these projects.

Documentation

Documentation seems to play an important part in OS programmers' information requests. Again, this was surprising because other 'information source' literature suggested that programmers distrust documentation (Singer 1998, Seaman 2002 and Sousa et al 1998). In our results however, *Documentation* was ranked the 7th most requested information focus across all projects in the dataset. Over all years of all projects, 8% of the questions were *Documentation* questions. Given the large number of total questions (739 questions from 2104 emails) in the dataset, this percentage suggests that documentation is an important issue for programmers (58 questions). Also, often these requests seemed very important for the information seeker: for example: "*Could anyone please tell me if Eclipse Platform is J2EE compliant, where could I get some more documentation on it. This piece of information is really critical for me*".

It is possible that the delocalized nature of OS programmers might be the reason for a higher-than-expected number of *Documentation* requests. As OS programmers cannot rely on informal, immediate communication with their team, they are more likely to need reference material in hand while doing their job. In addition, it is possible that, due to the delocalized context of programmers in this study, OS programmers may be motivated to produce better documentation, and therefore it is possible that the community trust documentation more than in the traditional case. These provisional findings suggest new research questions on how their working environment (their OS context) affects programmers' tendencies to use documentation as a reference in software maintenance.

Tools and Technology

Results shown in Table 5.3 also show a strong emphasize on *Tools and Technology*. Taking *IDE*, *Communication Channel* and *Operation Environment* as *Tool / Technology* based questions, 20% of the entire request-set, throughout all OS projects was for *Tool / Technology*. This suggests a huge focus of OS programmers is on the tools or technology that they currently use.

The high request for *Tool / Technology* might reflect the rapid changes in tools used by OS programmers. For example, a version of the Java Development Kit, namely Java SE 6, has had 6 updates released within 11 months in 2010 (Wikipedia, 2010). At the same time, Sun Microsystem also (during the time of study) was in the process of producing other versions of the JDK: namely Java SE 7.0 and Java SE 8.0. This rapid change is likely to impact on the programmers' coding and debugging work. One example that illustrates the environment of continuous change is: *“Do you remember what version of RELOAD was current, the time you dealt with it?”*¹³

OS programmers also might ask a lot of questions if the OS project chooses specific tool that they are not familiar with, such as *“Is Apache easier to deal with than Jakarta?”* The high request rate for this type of information might also be related with requests for software document. For example, there are questions in the dataset asking about the user manual for the specific tools they use: *“Is there any Apache official guidelines on this?”*

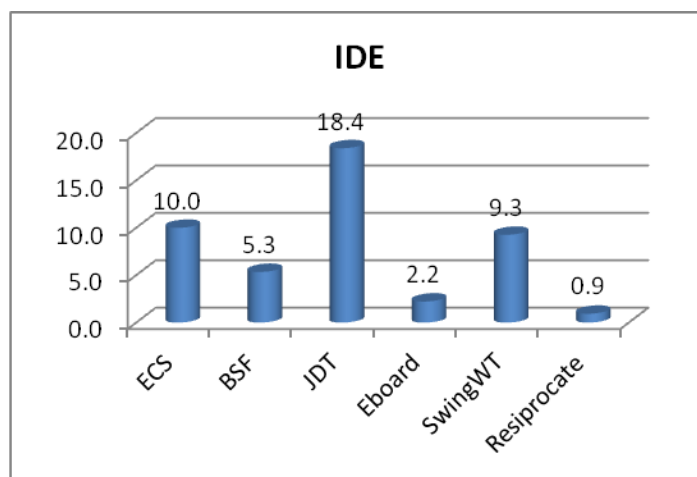


Figure 5.2 – Percentage of request for IDE focus in each individual OS project.

The *IDE* was the most commonly requested *Tools and Technology* category. As Figure 5.2 shows this was particularly so for the JDT project (18.4%). Since the JDT project is concerned with enabling Eclipse for Java development, the focus

¹³ RELOAD is an OS text editor.

of this Open Source endeavour is closely tied to the Eclipse IDE and hence we would expect a lot of questions about the IDE. This probably artificially raised the frequency of IDE-type questions.

Not-with-standing, it is possible that the high request rate for tool/technology information in general was related to newbie programmers who were familiarizing themselves with particular project tools and technology or new technology itself.

5.4.2 Information Aspect

The graph in Figure 5.3 presents the number of *Information Aspect* requests in the entire datasets along the Y axis. The categories of *Information Aspect* were ordered according to overall rank. Each line represents a specific OS project and these were presented in stacked fashion to illustrate the overall pattern of information request. Similar to Figure 5.1, Figure 5.3 below shows an (even more) consistent pattern of *Information Aspect* across all projects. The essential pattern seems to hold with small fluctuations across the *Who* and *Where* categories across different projects, suggesting generality of the ranking of the schema again.

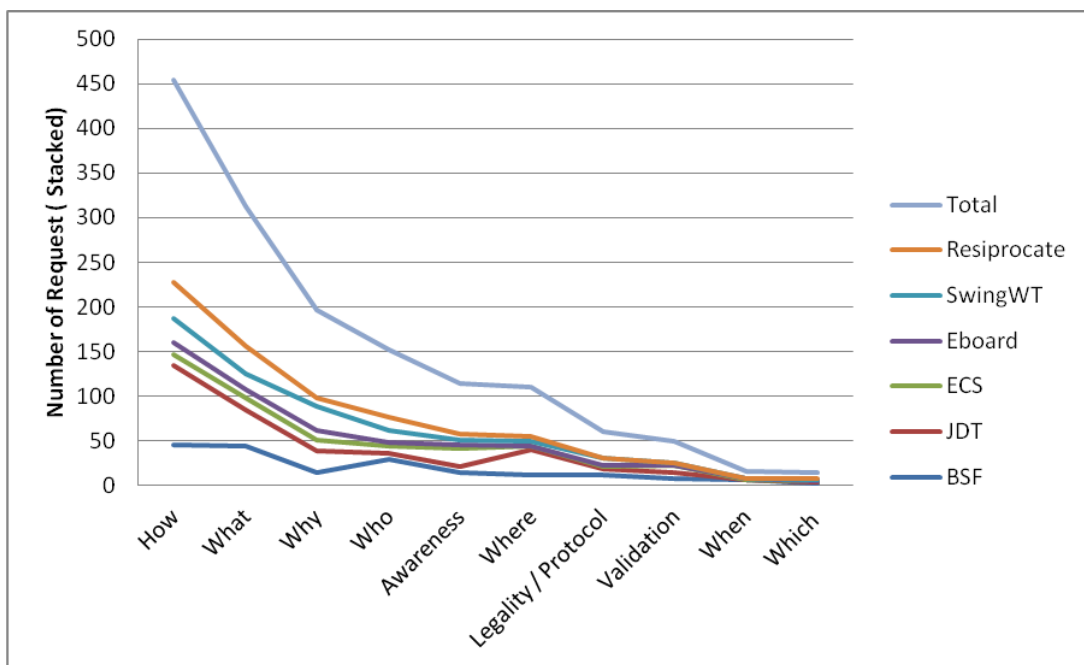


Figure 5.3 – Information Aspect for all dataset for this thesis.

How, What and Why

This graph in Figure 5.3 shows a predominance of *How*, *What*, and *Why* questions. This aligns closely with Letovskys (1987) findings which suggest lots of these types of questions in programmers debugging. Letovsky states that *How* questions are indicative of top-down approaches where the developer asks how the code, the system or a tool (for example) achieves its higher-level (already known) goal. *Why* and *What* questions are more bottom-up in nature where the programmer asks what the goal of a tool, system or code is, or why the low-level detail of these entities are this way.

Team-Awareness

While the majority of questions are *Why*, *What* and *How* questions, there is also a suggestion of an overall team orientation. That is, the *Awareness*, *Who* and *Validation* questions identified in the datasets seem to refer to the team-based nature of the development. In this context, *Awareness* questions reflect OS programmers' keenness to know what other programmers doing. *Who* questions request information on the member(s) of the team who (for example) have implemented a specific part of the system. Likewise the *Validation* questions ask others (inside or outside the team) for permission to take some course of action. Taking these three categories cumulatively as team-based information concerns, 21.4 % of the dataset were team oriented questions.

This is in line with Gutwin et al (2004), who suggest that distributed programmers need to maintain awareness among them, and that they maintain awareness of the entire team and more detailed knowledge of people they plan to work with. This is also similar with the results of a study on co-located programmers done by Ko et al's (2007). In their study, Ko et al (2007) found that programmers often need to maintain their awareness on certain information including task performed by the other co-workers.

Location-Based Question

There are also a considerable number of *Where* questions (7.4% of total requests).

The *Location* category of information wasn't present in early research that aimed to inform on the information types sought by programmers in the context of software comprehension (Good, 1999, O'Shea, 2006, Corritore and Wiedenback, 1991, Ko et al., 2007). These studies were done based on a theoretical analysis of the information available in programs, originally carried out by Pennington (1987a). In one typical example of this work, O'Shea (2006, 2004) did content analysis on one OS mailing list based on Pennington's schema and her resultant schema was heavily reflective of the original. Indeed, only late in O'Shea's work did she identify this new information types independent of Pennington's schema. She described '*location*' as an information type where programmers discussed the locations of fixes and functionalities in the code (O'Shea, 2007). This category wasn't present in Pennington's initial analysis, probably because Pennington only considered individual programmers studying small code pieces (Pennington, 1987a). However, O'Shea's finding is echoed in this work.

In a more recent example, Sillito et al (2008) studied questions that programmers ask when evolving source-code. They identified 44 types of questions asked by programmers and organized them into four levels of question categories (See chapter 2 – section 2.5.2). Nine out of 44 types of questions suggested by Sillito et al (2008) were explicitly location based questions.

Inline with O'Shea (2007) and Sillito et al (2008), the data in Table 5.4 shows a strong presence of '*Location*' aspect queries. 58 *Where* questions were identified in the datasets. This represents approximately 8% of all questions asked, suggesting that this is a significant information seeking aspect for OS programmers maintaining large systems. These finding also adds empirical credence to Rajlich's 'Concept Location' work (Marcus et al., 2005).

Process Orientation

Overall, OS programmers' information seeking is code oriented but it also has an emphasis on process. As shown in Table 5.6, taking *Stages*, *Distribution/Release* and *Support Required* as process related issues, 14.7% of the requests were process oriented. In the *Information Aspect* dimension (as shown in Table 5.7), *Who*, *Validation*, *Awareness*, *Legality / Protocol* and *When* questions directly reflect a process-oriented nature, while many of the *How* questions also reflect this aspect of software development. (E.g., “Does anyone know how one can file a bug report with Sun?”). Even disregarding the proportion of relevant *How* questions, 26.5% of questions are fairly explicitly process-based based on this analysis.

Table 5.6 - Process Related Issues of Information Focus.

Info. Focus	BSF	JDT	ECS	Eboard	SwingWT	Resiprocate	TOTAL
Stage	4	1	3	0	0	5	13
Distribution / Release	16	1	5	2	4	1	29
Support Required	28	8	8	4	8	11	67
Total	48	10	16	6	12	17	109
% of total request in the project	25.7	4.7	20.0	13.3	11.1	15.9	14.7

Table 5.7 . Process Related Issues of Information Aspect.

Info. Aspect	BSF	JDT	ECS	Eboard	SwingWT	Resiprocate	Total
Who	29	7	8	4	14	14	76
Awareness	15	6	20	4	6	6	57
Legality / Protocol	11	8	2	2	7	0	30
Validation	8	7	7	1	2	0	25
When	6	0	0	1	1	0	8
Total	69	28	37	12	30	20	196
% of total request in the project	36.9	13.2	46.3	26.7	27.8	18.7	26.5

These results suggest that this category of information seeking is significant for OS programmers and deserving of further study, given that most studies to date have not concentrated on this area (Sillito et al., 2008).

5.4.3 Information Focus & Aspect

Table 5.8 below presents the *Information Aspect* within each *Information Focus* derived from the dataset. Discussion of these results is presented in the following sections.

Figure 5.4 below visualises the figures presented in Table 5.8. It serves to direct investigation of clusters between *Information Aspect* and *Information Focus*. The top seven highest requests in the dataset are:

- *How* questions on *System Enhancement* (71 requests).
- *Who* questions on *Support Required* (59 requests)
- *Why* questions on *System Bug* (55 requests)
- *What* questions on *System Design* (37 requests)
- *How* questions on *Task Implementation* (36 requests).
- *How* questions on *IDE* (34 requests)
- *Where* questions on *Documentations* (30 requests)

In general, these figures reinforce the findings that OS programmers' information seeking is very implementation centric in nature.

Table 5.8 - Overall Analysis Result : Relationship between Information Focus Categories and Information Aspect Categories.

Focus vs Aspect	How	What	Why	Who	Awareness	Where	Legality / Protocol	Validation	When	Which
System Enhancement	71	11	10	1	7	3	0	2	0	1
Task-Implementation	36	18	2	5	19	4	6	13	1	0
System Bug	20	13	55	3	1	4	0	0	0	3
System Design	20	37	9	0	1	2	0	1	0	1
IDE	34	21	6	2	0	4	0	0	0	2
Support Required	2	1	1	59	3	0	0	1	0	0
Documentation	3	8	0	1	11	30	3	2	0	0
Communication Channel	11	5	6	3	2	2	9	2	0	0
Operation Environment	19	17	2	0	2	0	0	0	0	0
Distribution / Release	0	4	1	1	5	0	11	2	5	0
File Configuration	8	6	1	0	2	6	1	2	0	0
Task-Test	2	6	5	1	3	0	0	0	0	0
Stage	0	8	0	0	3	0	0	0	2	0

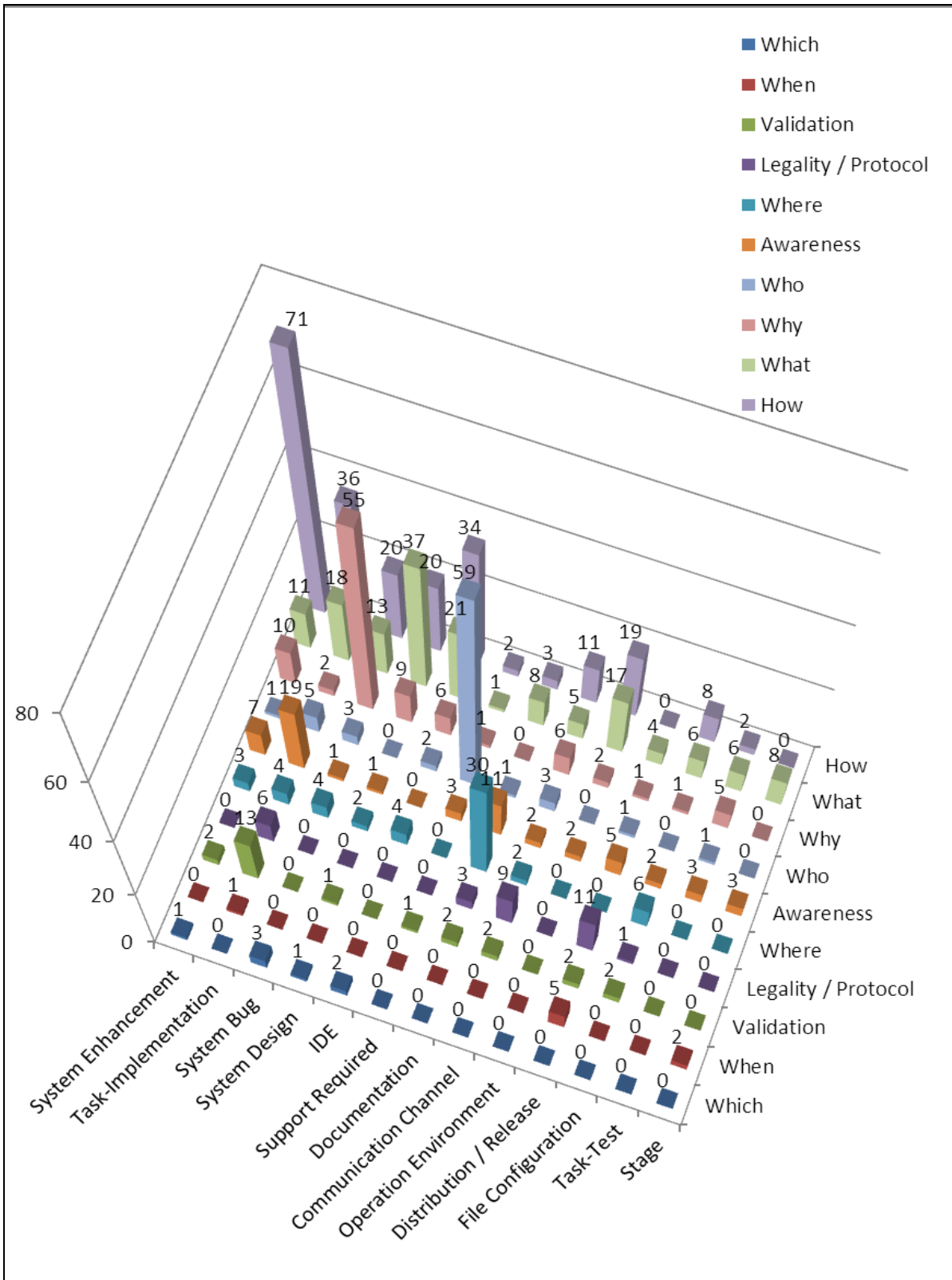


Figure 5.4 – Clusters of Information Focus and Information Aspect.

How* questions association with *System Enhancement* and *Task-Implementation

The high number of *How* questions on *System Enhancement* (71 request) and *Task-Implementation* (36 request) suggesting that programmers are keen to know how the code works (*System Enhancement*) and what steps should be taken (*Task – Implementation*) to achieve some system goals.

Request of this nature reflect a very top-down approach for OS programmers in enhancing their code (E.g. “*How can I further optimize the below chunk of code...*”). This suggests that the programmers asking these questions are trying to map their knowledge (about the program domain goals) to the source code (Shaft and Vessey, 1995).

Team Orientation

Results in Table 5.8 and Figure 5.4 reflect a strong team-oriented nature in OS programmers’ information seeking. Taking *Who* questions as directly acknowledging a team-based context, 59 *Who* questions were targeted on *Support Required*.

While a “*Who-Support Required*” category may reflect the increased effort in allocating and breaking down work in a delocalized context, Table 5.8 shows a more general underlying information need founded upon team awareness and team dynamics. There is high usage of *Awareness* questions in general, specifically on *Task-Implementation* (19 requests) and *Documentation* (11 requests). In the context of *Task-Implementation*, these reflect programmers’ concerns on tasks that are currently being done by other team members (Ko et al., 2007) such as, “*Robert are you still working on the texen stuff for ecs2 to generate the html and rtf classes?*”. In the delocalized context of OS programmers, this type of question seems important to coordinate task among OS programmers (as in “*I was wondering if anyone was working on this code, and if not, if anyone would mind if I took up an effort to finish it.*”). This is in line

with Bonaccorsi and Rossi's (2003) suggestion regarding the importance of good task coordination between OS programmers that working in parallel development.

Likewise, in a *Documentation* context, it reflects programmers concern to make sure that the other members in the team got the same document (For example "*I posted a concept to the news://news.eclipse.org/eclipse.tools, but I am not sure it would REALLY get read by this group in time to save tool development time(?)*")

Why questions association with System Bug

This relationship reflects programmers' reasoning about some errors (bug) found in the code (E.g. "*Dialog pops up and vanishes before I can read it - any ideas?*" or "*(Given an error situation)...I have no idea why this is happening. Please help me solve this problem.*" This is in line with Ko's (2007) findings on the information needs of co-located developers when he found that programmers spent lots of time in determining the reason for certain bugs or program states (Ko et al., 2007). It also suggests that a bottom-up comprehension approach was being used by programmers in their debugging. In this context, the '*Why-System Bug*' reflects programmers' efforts to map from the code to that bug-state.

What questions association with System Design

The high request of *What* questions on *System Design* (37 request) suggests that programmers tend to know about certain design decision. Examples include: "*What criteria are used for deciding where a particular BSF-engine should be a part of bsf or not?*", "*Is there any plan to add pattern functionality to the JDT UI*" and "*should BSF gain the ability to auto-load languages (requires discussion on this list) in the near future*". This is in line with Feller and Fitzgerald (2002) suggestion that, in OS setting, design stages are typically performed by single developer or small core group in advance. Hence *What* questions were asked because programmers were not involved during the design stage.

How questions association with IDE

This relationship (*How* questions on *IDE*) shows programmers' interest on how the IDE features are employed. For example, “*How can I just compile a subset of source folder within the project-one working set?*” and “*Is there any option of saving the bookmark for a project?*” This contradicts the suggestion made above that the project focusing on the JDK might have biased the number of *IDE* requests, suggesting instead, a genuine need for a user manual of the *IDE* used in the project. This might be related then with the prominent relationship between *Where* questions and *Documentation*, as discussed below.

Where questions were very associated with Documentation

The high request rate (30 requests for information) for *Where* questions with a *Documentation* focus shows that most documentation requests wanted to find *where* the documentation was, rather than *what* was in it. This suggests that OS programmers know that their desired documentation exists but do not know *where*: “*alternatively if you can point me to instructions on how to do it I'll be happy to do it.*”).

5.4.4 Trend over OS Project Category

Figure 5.5 presents percentage of implementation centric focus requests for all OS projects in the dataset¹⁴. According to Daniel (2009a), Eboard, SwingWT and Resiprocate are considered successful and have long maintenance life-spans. Hence these three projects report on a community that potentially has active maintenance activities. On the other hand, ECS is an abandoned project, based on very low activity in its email communication. The datasets for this project shows that, it has only 398 emails with 80 questions within an 8 year period. According to the ECS project website (<http://jakarta.apache.org/ecs/>), the

¹⁴ Percentage shown in the diagram representing request rate in each individual OS project.

project has been retired since September 2010 due to the lack of development activities, and so can be considered unsuccessful. Finally, BSF and JDT are considered as between successful and non-successful. These projects are referred to as intermediate-success projects in this thesis.

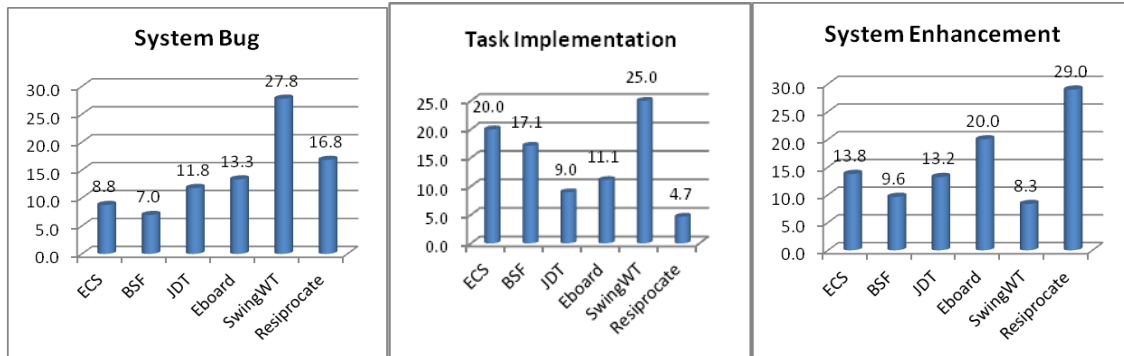


Figure 5.5 – Percentage of Request for implementation centric focus over all OS project.

Overall, the figures in Figure 5.5 show that all three categories are requested quite frequently in all projects.

System Bugs

In analyzing the percentages of request (Figure 5.5), one of the interesting issues is the number of requests focused on *System Bugs*. As shown in the figure, this focus was ranked as the most requested focus of the implementation categories in successful projects. One possible reason for this might be because these projects have long maintenance life spans and an active user base. Consistent with our interpretation of a driving user-base, the requests for *System Bug* information was lower in BSF and ECS: less and non successful projects.

Interestingly these findings align roughly with (Swanson, 1976) who suggests that only 17% of maintenance is *corrective* and that there is a stronger emphasis on *perfective* (61%) maintenance (*System Enhancements* in Figure 5.5).

Task Implementation

One of the interesting points illustrated in Figure 5.5 is the high request for *Task-Implementation* in the abandoned ECS project. 20 % of requests in the ECS project were directed at *Task-Implementation*. Figure for this focus were slightly lower among intermediate-success project (BSF and JDT), although the BSF project was still high in this regard, and the figures fluctuate between successful projects (Eboard, SwingWT and Resiprocate).

The high focus of this on the abandoned OS project ECS seems very strange as it suggests that tasks are being done in that project. However, this is largely a result of the smaller data-set in the ECS project. On closer viewing there were only 16 requests of this type (out of 80 questions for all focus) during the 8 years archive of the dataset (on average, 2 requests per year), suggesting very low implementation task activity in this project. By comparison in the BSF project which seemed to have a lower percentage (17%), that percentage actually reflected a much greater number of requests (32 request).

System Enhancements

Another interesting issue was the high request rate for *System Enhancement* information among successful projects, again suggesting an enthusiastic community. This focus was highly requested with 29 % of requests for Resiprocate and 20% in Eboard project. While it is hard to rationalize the lower rate of request in SwingWT (8.3%) without further qualitative study, the possible reason for the high request in Resiprocate and Eboard would seem to be ongoing evolution that demands ongoing understanding of the system, and possibly the incorporation of new programmers.

Documentation

An interesting finding, shown in Table 5.3 is the significant request rate on *Documentation*.¹⁵ In analysing this focus, Figure 5.6 below presents the rate of request (in percentage) for *Documentation* over all the OS projects. As discussed in the previous section, Eboard, SwingWT and Resiprocate are communities that potentially have on-going, active maintenance. On the other hand, ECS is an unsuccessful project and BSF and JDT are of intermediate-success.

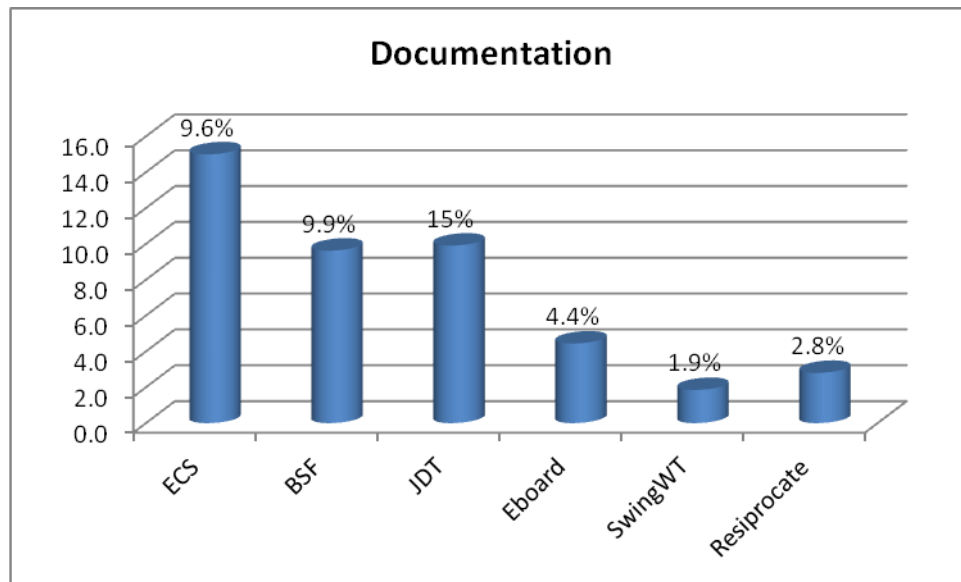


Figure 5.6 - Request for System Documentation Focus over all OS projects.

Overall, Figure 5.6 shows that the request for *Documentation* is comparatively low in successful OS projects and gets higher as the success of the project decreases. This is an interesting finding and difficult to interpret.

However, Daniel et al (2009a), does state that successful OS projects often provide documentation on their website to encourage development (Daniel et al., 2009a). Hence, the requested documents would be easily accessible by programmers and consequently this would reduce the number of request in successful OS projects. Conversely, he suggests that documentation may not be available on the website of unsuccessful OS projects.

¹⁵ Percentage presented in the diagram referring to number of request for Documentation focus in each individual OS project.

Further investigation was done on the three projects that were reported as having high requests rates for documentation. Investigation on the ECS and BSF project's website shows that the documentation provided was very limited. However, investigation on the JDT project's website during September 2011 (during this writing) shows that comprehensive documentation was provided and was easily downloadable by programmers. Given the finding that documentation was highly requested in the JDT dataset, this website seems to contradict the proposed rationale. However, it is possible to rationalize this anomaly with respect to the timeframe of the dataset. The datasets for JDT were taken from the first 3 years of its evolution (2002, 2003 and 2004). It is possibly that during this time, there was a lack of documents on their website. The document sharing may have happened after 2004, as a respond of programmers' requests. It should be noted though that this is only a hypothesis and would need further investigation for confirmation.

Hence, a plausible conclusion is that the number of documentation requests is affected by the availability and the accessibility of documentation among OS programmers. This suggests that OS projects should provide programmers with appropriate documentation on their website or in any communication channels they use. The documents shared in the website should be updated according to any changes in their projects.

5.5 Responses to Information Request

The literature shows two opposing perspectives that can be related to OS programmers' behaviour in responding to questions. OS programmers have been generally characterized as communities with highly proactive and motivated contributors in the past (Feller and Fitzgerald, 2002, Gacek and Arief, 2004). However, software maintenance is often portrayed as an undesirable task, that programmers tend to shy away from (Sommerville, 2004).

In the context of this research, response analysis was carried out to evaluate the two opposing perspectives mentioned above and see if the pro-activity associated with OS development overcomes the reticence of programmers with respect to maintenance, at least in their support of other programmers in the community. Specifically, response analysis was done to investigate the likelihood that each OS programmer would receive responses and the amount of responses they are likely to get. For this purpose, response analysis was performed on all the questions found in the dataset.

This analysis work involved tracing all emails that responded to the original emails, counting the number of responses and recording the time when the original emails (asking question email) and responding emails (first and the last email) were sent. All of the information was then recorded in the spreadsheet along with the related questions.

Table 5.5 represents analysis of the responses received for the most popular query types posted by the OS programmers on the mailing lists¹⁶. The first column reports on the information sought (its focus and its aspect). Column two reports on the number of queries identified for each information-type and column three presents this as a percentage of the whole. Column four shows the percentage of these queries that received a response and column five reports on the average number of responses received, for queries that received at least one response. Finally column six reports on the average number of days which passed between the query being posted and the final response, again for those queries that received at least one response.

¹⁶ Table 5.5 (as previously presented in section 5.3) is redisplayed here to help discussion in this section

Table 5.5 (replicated from earlier) – Response Analysis Result for Information Focus and Aspect Dimension

Information Focus	Total No of Question	% of Total Request	% Answered	Average No of Response	Average Timespan of Response (days)
System Enhancement	106	14.3	64.2	2.4	1.5
Task - Implementation	104	14.1	63.5	2.8	2.9
System Bug	99	13.4	55.6	2.4	3.0
System Design	71	9.61	69.0	2.6	2.8
IDE	69	9.34	62.3	2.4	1.6
Support Required	67	9.1	55.2	2.8	3.6
Documentation	58	7.8	58.6	2.6	3.4
Information Aspect					
How	227	30.7	58.6	2.3	3.3
What	156	21.1	59.6	2.7	1.6
Why	98	13.3	67.3	2.2	1.9
Who	76	10.3	60.5	2.8	3.3
Awareness	57	7.7	57.9	2.9	1.9
Where	55	7.4	60.0	2.8	4.1
Average			61.3	2.5	2.5

5.5.1 Low Response Rate

Perhaps the most surprising finding is the low response rate overall. On average, a query had around 61% chance of being responded to: less than a two-thirds chance. Among the top 3 most requested *Information Focuses* (see Table 5.5) the lowest response rate was for *System Bug*. Indeed, when the programmers were interested in seeking information on *System Bugs* they had only a 50:50 chance of getting a response. As this is among the most frequently sought information focus, this is problematic for the community. Popular *Information Aspects*, with a low success rate include *Awareness* questions and *How* questions, which together make up over 38% of all the questions posed. By far the highest response rate was recorded for *System Design* queries. 69% of these queries were responded to by the community.

These figures are slightly surprising, especially for this technical, OS community. Given, the fact that most OS programmers are considered highly motivated

developers (Feller and Fitzgerald, 2002, Bonaccorsi and Rossi, 2003), they might have been expected to have higher response rates.

Low Response Rate of *System Bug*

It is difficult to hypothesize on the reasons for these response rates without in-depth qualitative analysis. However, as shown in Figure 5.7, most of the questions with *System Bug* focus were directed at *Why* (56%) and *How* (20%) aspects. '*System Bug - Why*' questions are considered difficult (Ko et al., 2007) and responders might shy away from such questions. Likewise, '*How-System bugs*' also appear difficult: This requires reasoning about systems' behaviour when exhibiting a bug. So, the answer itself may take some time and effort to formulate, thus driving potential responders away.

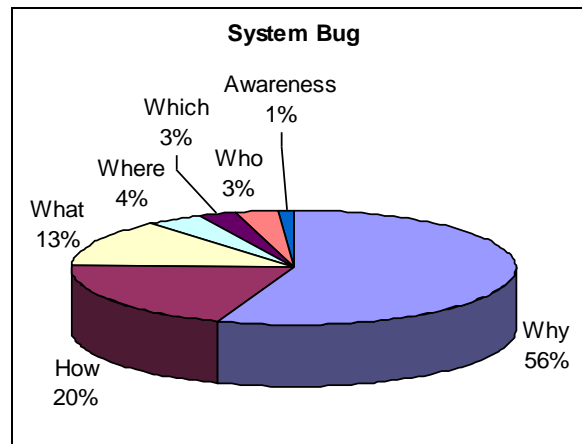


Figure 5.7 – Info Aspects of System Bug

Low Response Rate of *Support Required*

Another *Information Focus* with a low response rate shown in Table 5.5 was *Support Required* with only 55.2 % responses. However, as shown in Figure 5.8, 89% of the questions on this focus were directed at *Who*. In the context of *Support Required* the response of *Who* question (such as, “*Can anyone commit this testcase for MethodUtils class.*”) reflects the willingness of OS programmers in helping their team members and this low response rate might initially suggest a very reluctant community. However it is likely that such requests can only be

answered by a few specific people who are available or have the ability to commit to the requested task, suggesting a small pool of programmers to answer the question.

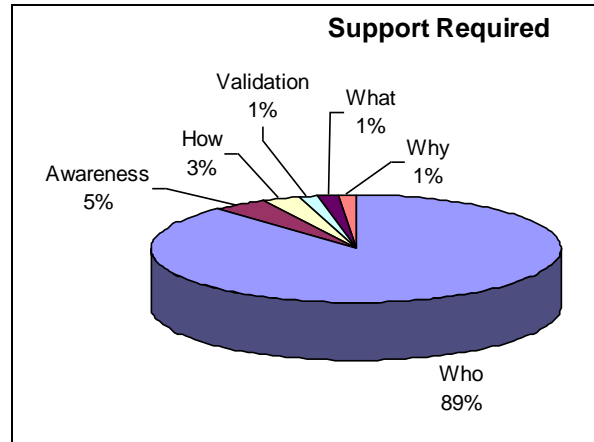


Figure 5.8 – Info Aspects of Support Required.

Low Response Rate of Tools & Technology Based Questions

As shown in Table 5.3 (section 5.3), there was a strong emphasize on *Tools & Technology*. Taking *IDE*, *Communication Channel* and *Operation Environment* as *Tool & Technology* based question, 20.2 % of all information requests were on *Tools & Technology*. As shown in Table 5.9, the response rate for this type of questions was below average (60.4%).

Table 5.9 – Response Analysis Result for Tools & Technology Based Information Focus.

Info Focus	Total No of Question	% of Total Request	% Answered	Average No of Response	Average Timespan of Response (days)
IDE	69	9.3	62.3	2.4	1.6
Communication Channel	40	5.4	55.0	1.9	1.4
Operation Environment	40	5.4	62.5	2.5	4.7
Total / Average	149	20.2	60.4	2.3	2.4

However, on closer viewing, the low response rate for *Tools & Technology* in general was highly contributed to by the rate of response for *Communication Channel* type questions. While *IDE* and *Operation Environment* got above average response rate (more than 62%), the response rate for *Communication Channel* was far below average with only 55% responses. A possible explanation

can be suggested by looking into the predominant *Information Aspects* around the *Communication Channel* questions. As shown in Figure 5.9, most of the *Communication Channels* questions were on *How* (27%) and *Legality/Protocol* (22%). In the context of *Communication Channel*, *How* questions reflect requests for guidance or instructions on using facilities in the channel to achieve the programmers' goal. (for example, "I can't login to the wiki .. how can I create an account ?"). *Legality/Protocol* questions refer to the programmers request about rules that related to specific *Communication Channel* (such as, "Is this the right mailing list (for developers)?"). These types of questions can presumably be answered with ease by experienced programmers. But it may be that potential responders feel that this knowledge is passé and that contributors should know about the project's communication channel in advance. Thus they may be reluctant to help them for this *'technical-snobbery'* phenomenon, where OS programmers expect community members to acquire the required and possibly basic knowledge about their project surroundings. An alternative hypothesis, related more specifically to a subset of the *Legality/Protocol* questions (such as, "could we get jira@apache.org allowed to post to the dev list so the JIRA notifications get through?"), is that such questions could only be answered by team leader or in charge person; hence limiting the audience to answer the questions. However, these are only hypothesis and would need validation by other researchers.

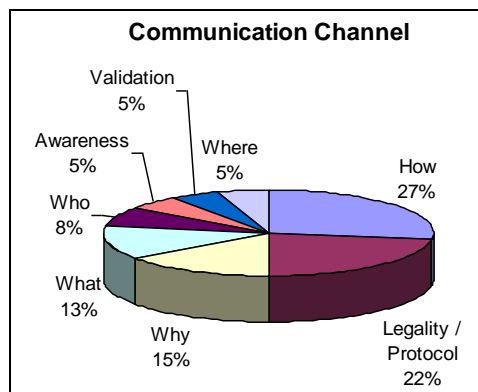


Figure 5.9 – Info Aspects of Communication Channel Info. Focus

Low Response Rate of Documentation

Another *Information Focus* with below average response rate shown was *Documentation* (58.6%). In analysing the *Information Aspect* sought around *Documentation* (Figure 5.10), it shows that most of the requests on this focus were *Where* (52%) and *Awareness* (19%) aspects. Such questions don't seem to require complex explanation and can often be easily answered. However, answers for *Where* (such as in, "where could I get some more documentation on this.") and *Awareness* (such as "I don't seem to be able to find the document. Is that me missing some mails or was it forgotten?") questions are likely to be related to the availability and the accessibility of documentation among the OS programmers themselves. It suggests that, when developers ask this type of question the documentation isn't available to them and so may not be available to the other developers either. This suggestion is reinforced by the number of requests for documentation external to the actual development project itself (for example, there is a question that asking about memory management in Eboard project which is not directly related with the project but might be useful for programmers: "Does anybody know of some resource on memory management with GTK+ (article, online book, tutorial)?"). Meaning that, even if project documentation is available on the homepage, it may not support this information need.

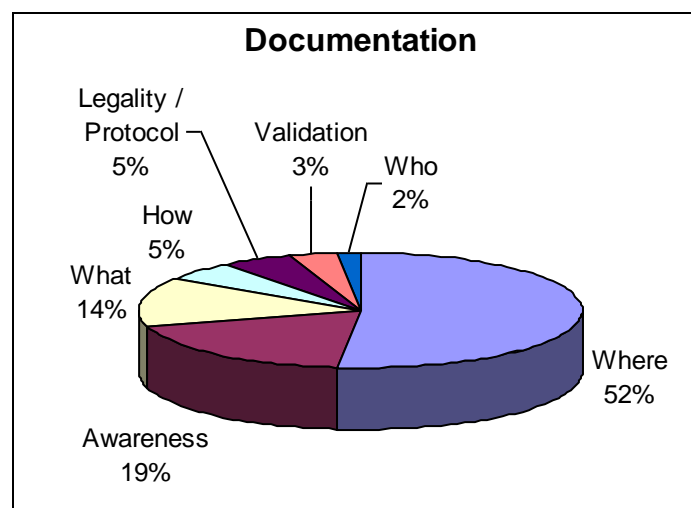


Figure 5.10 – Info Aspects of Documentation Info. Focus

Low Response Rates among Information Aspects

Table 5.5 also shows that the lowest response rates among *Information Aspects* were *Awareness* (58%) and *How* (59%). While the low response rate for *How* questions can be reasoned as per the rationale in the previous paragraphs, it is difficult to explain the low response rate to the *Awareness* question in this fashion. Such questions should not be that difficult for the technical community and probably require a short answer. In depth analysis on *Awareness* questions, as shown in Figure 5.11 below, show that most of the questions were on *Task - Implementation* (33%) and *Documentation* (19%). The rationale for low *Awareness* requests for documentation was discussed above. But *Task Implementation* type questions often ask about latest update of something to make sure that the requesters have the latest information that relates to their task. It is possible though that these questions only can be answered by the specific person in charge of the requested subject.

This finding suggests that OS project groups should consider facilitating their community members with organizational chart (to discuss 'who is doing what') and sufficient document should be provided that reach beyond their project, on their website or any communication channel that they normally use.

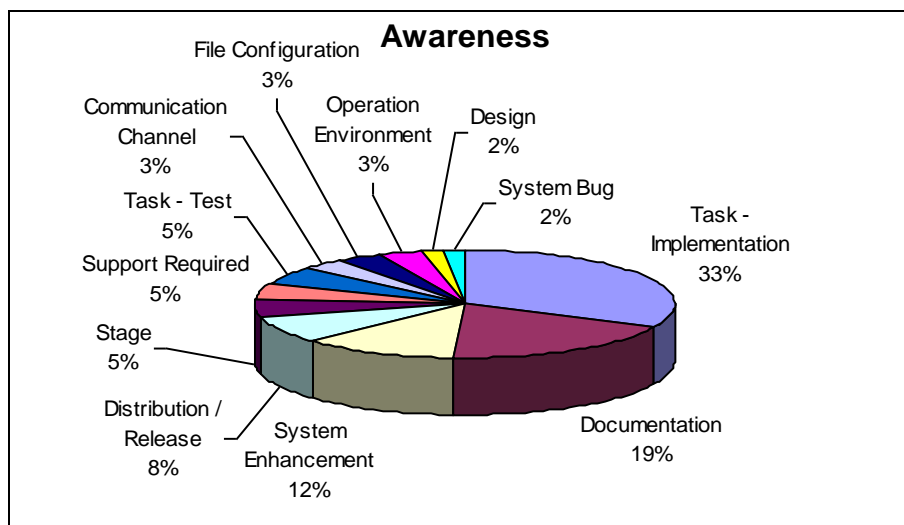


Figure 5.11 – Information Focus Categories for Awareness Aspect

5.5.2 High Rated Information Types

Among the highest response for *Information Focus* was *System Design* (69%). The possible reason for this rate of response is that design phase (in OS setting) is performed by a single programmer or small group of core programmers (Feller and Fitzgerald, 2002) that feel ownership of (and thus pride in) the project. Hence, to make sure that later contributors (OS programmers) are developing the right program (as per their design), these designers might have a tendency to share the information.

The highest response for *Information Aspect* was *Why* questions (67%). This is surprising as *Why* questions seem to require a more difficult explanation (Ko et al., 2007) and were proposed as the reason why programmers were adverse to answering *System Bug* questions. If anything, this finding suggests that for *System Bug* type questions, programmers seemed to be less likely to respond based on the fact that they were *System Bug* questions rather than *Why* questions.

5.5.3 Discussion on Answered Questions.

Regarding the questions that were answered, they did seem to provoke some discussion, resulting in an average of 2.5 responses for each (responded-to) query. This suggests a discursive community, particularly with regard to *Awareness* questions (where the average number of responses, when answered, was 2.9). This is surprising as discussion implies a degree of animation in the community and questions such as '*Robert are you still working on the texen stuff for ecs2 to generate the html and rtf classes?*' would not seem, on first impression, to have the contentiousness to stoke up such animation. Again, further qualitative analysis is required to probe this finding.

Discussion also happened around *Task-Implementation* and *Support Required* type questions with 2.8 responses for each query. The figure for *Task-Implementation* reinforces the suggestion of implementation centric behaviour in

OS programmers information seeking. Specifically it suggests that the community are happy to discuss implementation level concerns. Likewise, figures for *Support Required* reinforces the suggestion of process orientation nature in OS programmers' information seeking. This suggests that OS programmers are happy to discuss the support they might offer (only if they are able to help, given the low response rate for *Support Required* discussed earlier).

The least amount of discussion happened around *Why* questions (2.2 responses on average) and *How* questions (2.3 responses on average). Ironically, *Why* questions also gained the highest response rate among all information aspect. It is very hard to tally this situation although it does suggest that there is a limited pool of people with this knowledge that are prepared to answer these questions. That is, *Why* and *How* questions seem to require a detailed explanation, leading to verbose, time-consuming answers. It suggests that only programmers that have the information (in this context experienced programmers) to answer the question will give their response, while the rest will be less likely to join in the conversation.

This finding could also be related with the low discussion rate for *System Enhancement*, *System Bug* and *IDE*. On average, these information focuses only got an average 2.4 response rates. Most of the questions for these information focuses were on their *Why* and *How* aspects. Hence, it can be hypothesized that the limited discussion was related with their aspect rather than their focus. Again, these are speculations only and would need to be probed by more in-depth, qualitative analysis of the community.

5.5.4 Response Time-span.

The largest distribution in the results set is in the response time-span (see Table 5.5, column Average Time-span of Response), which spreads from 1.5 to 3.3 days. Interesting in this regard is the relationship between this data and the data in column 4: Average Number of Response. For example, even though *Awareness* questions had a large number of responses (2.9) to each query, they

had a below average time-span (1.9 days), suggesting that the responses came quickly after the original questions were posted. In contrast, *How* aspect queries had a below average number of responses (2.3), but over a much longer time-span (3.3 days). Likewise, *System Bug* focus gained below average number of respond (2.4) within 3.0 days. This phenomena, suggests a different response-rate for different types in information retrieval.

5.6 Conclusion

This chapter compiled all the datasets used in the formation of the schema. All the extracted questions were analyzed based the resultant Information Seeking Schema. In doing this, we found both expected and surprising results. Specifically, it found that, in line with other studies, OS programmers were implementation centric. They often required location information and they were quite team-oriented. Another finding was that they tended to rely on documentation more than previously reported for non OS programmers, possibly due to their delocalized OS context.

Surprisingly, there was a focus on *Tools and Technologies* and *Process-Based* questions. There was also a surprisingly low overall response rate to their queries from the community, particularly with regard to a *System Bug focus*, a cause for concern, given its prevalence. Great discussion focused round *Awareness* type questions and this discussion typically happened quickly. One of the least discussed aspects was *How* questions but discussion of these questions happened over a much longer time-span. Further empirical studies need to probe the rationales behind this phenomenon.

The result of response analysis also suggest a ‘technical-snobbery’ phenomenon that might occur amongst the OS programmers’ community where *Why* questions were answered frequently but *Communication Channel* questions were answered infrequently.

These findings suggest a number of directions for IDEs and visualization tools that support programmers involved in OS developments. Details on the implications of these findings will be discussed in the concluding chapter.

Chapter 6

**Conclusion and
Future Works**

CHAPTER 6 Conclusion and Future Works

6.1 Introduction

This chapter presents the contributions and findings of the thesis, resultant from the thesis questions in Chapter 1. Following this, consideration is then given to the direction of future work.

6.2 Contributions

The major contributions of this thesis are as follows.

- Identification of the need for information seeking studies of distributed open source programmers, working on a large and complex OS system.
- Development of an analysis schema for determination of the information sought by OS programmers during OS software evolution.
- Insight into the information artifacts probed by OS programmers through their mailing lists, during OS software evolution.
- Insight into the type of information sought within the probed artifacts by OS programmers during OS software evolution.
- Analysis of the mailing list response rate for OS programmers' information requests during OS software evolution.
- Implied recommendations for IDEs, Information Seeking Support (ISS) tools and visualization tools that support programmers' information needs in OS developments.

6.3 Revisiting the Research Questions

The original thesis questions from Chapter 1 will now be revised with a brief discussion of the results. This thesis aims to characterize information seeking in open source software projects in term of the:

- a. Information artifacts probed by programmers.
- b. Information sought within the probed artifacts.
- c. Responses gained from the open source programmers' information seeking activities.

In addressing these questions, analysis on 739 questions extracted from 2104 OS programmers' emails, over several OS systems has been performed, as reported in chapter 5.

6.3.1 Information artifacts probed by programmers

The analysis result suggests 13 information focus (artifacts) identified in OS programmers' information requests as shown in Appendix B - Table B1. Analysis on the information focus probed by OS programmers suggests several interesting findings:

- OS programmers' information seeking activities are implementation centric. That is, much of the information seeking was directed at the implementation of the system with a strong code focus: *System Enhancement*, *Task-Implementation*, and *System Bug*. This is in line with existing literature in the area (Sousa et al 1998, Singer et al 1998).
- OS programmers' information seeking has a strong emphasis on *Tools and Technology*. Taking *IDE*, *Communication Channel* and *Operation Environment* as *Tool / Technology* based questions, 20% of the entire request-set, throughout all OS projects was for *Tool and Technology*. The lack of reference to this in previous literature illustrates that, for OS programmers at least, the effort lies, not just in evolving the systems but in

the increasingly complicated infrastructure surrounding the evolution of these systems - a novel finding.

- OS programmers' information seeking focuses more on *Design* than the previous literature suggests.
- OS programmers rely more on documentation than previous empirical studies of non OS programmers would have implied. *Documentation* was the 7th most sought *Information Focus* in OS programmers' information request.

6.3.2 Information sought within the probed artifact

There are 10 aspects of information sought within the probed artifacts mentioned above. This is shown in Table B2, Appendix B). Several interesting findings are suggested from the analysis of *Information Aspect*.

- OS programmers often using top-down and bottom-up queries in their evolution work (e.g., when working with code, systems and tools). This is shown by the predominance of *How*, *What*, and *Why* questions in their information requests.
- OS programmers are quite team-oriented in their questioning, possibly as a consequence of their delocalized context. They ask for *Who*, *Awareness* and *Validation* information within the probed information artifacts.
- OS programmers often required location information. This is evidenced by the *Where* questions in their information requests.
- OS programmers also emphasize process queries. Taking *Who*, *Validation*, *Awareness*, *Legality / Protocol* and *When* questions as reflecting a process-oriented nature, 26.5% of questions are fairly explicitly process-based. Again, this is a novel finding.

6.3.3 Clusters of Information Artifacts and their Information Sought

The investigation of relationships between *Information Focus* and their *Information Aspect* shows several predominant relationships between the two dimensions that serve to reinforce the above findings:

- Programmers are keen to know how the code works (*System Enhancement*) and how a task should be approached (*Task – Implementation*) to achieve some system goal. This is shown by the high number of *How* questions on *System Enhancement* and *Task-Implementation*.
- As would be expected, high numbers of *Who* questions were targeted on *Support Required*. This team-support agenda is also reinforced by the high number of requests to maintain their awareness on the current tasks done by other programmers (*Task-Implementation*) and current *Documentation* that is related with their task.
- Programmers often reason about the errors (bug) found in the code. This is shown by the strong association of *Why* questions with a *System Bug* focus reflecting Ko's findings for co-located programmers (Ko et al., 2007).
- Other interesting aspects include programmers desire to know *What* the *System Design* is, *How* to employ *IDE* features and the location of *Documentation*.

6.3.4 Responses gained from the open source programmers' information seeking activities

Results of the response analysis on each question in the mailing lists shows that the overall average response rate for OS programmers' information requests is lower than might be assumed, given the general assumption about the high motivation quotient of OS programmer communities that has been reported by

several researchers in open source area (Feller and Fitzgerald, 2002, Gacek and Arief, 2004).

On average, a query had only a 61.4% chance of being responded to. This suggests the needs for Information Seeking Support tools for programmers.

Analyzing the amount of responses gained by each of the information categories in the schema informs on the unanswered information requests-types that OS programmers face. For example the response rate for *System Bug* information and *Communication Channel* were very low (55.6%). Given that *System Bug* was one of the most requested information focuses, the low response for this type of question should be considered problematic for OS communities.

6.4 Implications of this work

The exploratory study reported here has implication in academic and in industrial contexts. In an academic context, the findings reported here can be used to drive further research agendas in the information seeking area. Specifically, the information seeking schema suggested by this study would probably be useful in further empirical studies of information seeking, as an encompassing framework for data analysis. During the application of the framework, it could be expanded and refined by other researchers. For example, the information seeking schema could be used for further analysis with different sampling criteria, such as code base questions, to further refine the code-specific information types sought by programmers. This would produce more detailed categories of *System Enhancement* and *System Bug* questions. The schema also could be applied to analysis of data taken from commercial development settings, to see if any different information seeking behaviours arise between OS and non OS / commercial software development.

The specific findings here could also be useful in defining research in the software comprehension area, the software visualisation area and the Concept Location area. Specifically, it suggests software comprehension research should

focus on what programmers really try to understand (for example system bugs, and how the code works). In other words, research in the software comprehension area should focus on the problem solving perspective of real programmers (Détienne, 2002).

Consequently, it also identifies information that should be visualised: The application of the schema gives us a ranking of the information that should be visualised for OS programmers on each project. For example this research suggests that the information to be visualized should be *System Enhancement*, *System Bug* and *Design* information. Finally, the research suggests further research in software visualisation that supports these agendas and Rajlich and Wilde's (2002) Concept Location agenda.

In an industrial context the findings reported in this thesis suggest a number of directions for IDEs, ISSs and visualization tools that support programmers involved in OS developments. Specifically, as mentioned above, it suggests a ranked list of information artifacts (focus) and their related aspects that are often sought by programmers. First and foremost it suggests that the community's continued focus on implementation is valid and valuable with respect to OS development.

Additionally, it may be able to offer guidance for the development of a web portal for on-line communities that supports open source software development. This is in line with literature from the Computer Supported Co-operative Work (CSCW) arena that suggests the needs of software to support collaboration (Shen et al., 2008). In the context of such collaboration in OS development, the findings of this study suggests that CSCW tool should support programmers' need to maintain awareness among team members about their working surroundings and the changes the team have made. It also suggest that organizational charts be made available to inform new developers of the roles of other developers in the community.

In the context of tool development, the high request rate for *System Enhancement*, *System Bug* and *Design* information suggest the importance of

visualising information related to incremental change, debugging and related design decisions for specific features of software under development. Specifically, the tools should be able to visualise top-down processes to support incremental change, as well as visualisation of bottom-up processes to support debugging activities. Such software should also visualise the mapping between the problem domain (concept) and the solution domain (the source code) to support the concept-location approach.

Another important suggestion is that documentation should support programmers in maintaining their awareness of their working surroundings. Likewise tutorials should be made available on tools/technology that the community uses during software evolution, as this seems like a significant information need for developers.

6.5 Limitation of this research study

The study reported in this thesis was designed to employ Grounded Theory and content analysis. However, in implementing the methodology, some possible limitations were noted:

i) Reliability of the derived schema

The information seeking schema was directly derived from observations of the information types that programmers seek in-vivo. However, this could be biased since it was generated largely from the observations of the author and discussions with the research supervisor only.

However, in line with the Grounded Theory method employed in this study, the schema was “brought up” from the data and free from any potentially constraining theoretical harnesses. It was derived using very transparent and largely accepted protocols (Grounded Theory and Content Analysis). Additionally, the resultant schema has a high overlap with the previous literature. Specifically in the *Information Focus*

dimension it is in line with other research such as Sousa et al (1998), Singer (1998), (Seaman, 2002) and Ko et al (2007), In the *Information Aspect* dimension there is a high overlap with the standard 5WH framework of questioning.

ii) Communication channels used by OS programmers.

The mailing list was chosen as the observation medium in this study because it was suggested as one of primary communication channels for OS programmers (see chapter 2 – section 2.6.3). However, there are also another communication channels that could possibly be used by OS programmers: discussion forums, chat systems and social network applications (e.g. Facebook and Twitter). Hence, there is potential that this research could have missed important communication channels for OS programmers.

iii) Sample size.

While efforts have been made to achieve saturated data (as discussed in chapter 3-section 3.7), it is still possible that the dataset is insufficient for this purpose.

The dataset used in this study was taken from six mailing lists that consist of 17 years archived communication. In the grand scheme of things this dataset might still be considered small. For example, we noticed that the dataset used in this study was insufficient to explore all the emergent issues that arose during the study. Hence, there were many issues that were hypothesized in the research (such as the drip effect and the technical snobbery phenomenon suggested in chapter 5) that require more theoretical sampling and detailed qualitative analysis.

Another related issue was that this study focussed more on successful projects when considering Daniel et al 's (2009a) OS project categorization. Mailing lists from successful projects were chosen as they suggest longer maintenance phases and thus have the potential to contain

more maintenance activities (see chapter 2-section 2.6.4) and provide more useful information. However, after reflection, it is noted that this study should also have focused on the unsuccessful projects as the analysis might have provide more guidelines for potentially saving these projects.

iv) Programming language used in OS project.

Most of the datasets were taken from the Jakarta project website due to the reasons mentioned in section 4.2. As a result, all OS projects involved in this study used Java. Given the studies that suggest different programming language might influence programmers' cognitive process while working with code (hence affecting the information seeking behaviour Green (2000)) this study should be extended to other popular programming languages like C++, and C#.

Given the fact that this study is exploratory study in nature, the limitations mentioned here only partially affecting the current findings. Instead, they suggest unexplored emerged issues that prompt further exploratory studies and thus provide further research direction.

6.6 Future work

The schema generated in this research can be adopted, evaluated and refined by other researchers in the field to assess the generality of the results provided here. In doing so, the schema will evolve and could become established as a resource for all researchers in the area.

One particular analysis that was not done as part of this research was assessing the trend of OS programmers' information seeking over time. The dataset used in this thesis was not appropriate to investigate trends apparent in OS programmers' information seeking over time due to reasons such as the different archive time-scales used. A future study should consider extracting a longitudinal

dataset from several successful OS projects with a standardized time-scale across projects to probe the changes in information needs over time (e.g. a 10 year email archive for all involved OS project). This is potentially very interesting, given the constant stream of new programmers that are incorporated into OS projects, and intuitions about the different sorts of maintenance that are undertaken over the evolution lifecycle of systems.

There are also several other research directions to continue this work. A few examples of potential research directions arising from this thesis include:

- Defining appropriate visualizations based on the programmers' information need. For example, as discussed in section 6.4, visualisations that relate to system incremental change, system debugging, and their related design.
- Defining the software documentation that provides better support for programmers' information seeking and perhaps programmers' comprehension and Concept Location processes.
- Defining the requirements for ISS tools that will help programmers in obtaining all their required information and maintaining awareness about their project / task surroundings.

As mentioned in section 6.5, this research will continue to investigate the emergent issues that arose during this study. These issues could be investigated by qualitative studies that assess:

- How the working environment differences of programmers seem to affect their dependency on documentation?
- The possibility of a *Drip Effect* and *Technical Snobbery* in OS programmers' information seeking behaviour.
- The reasons why, various information seeking issues might decrease /grow overtime.

- The factors at play for high or low response rates for particular types of queries like *Communication Channel*, *System Bug* and *System Requests*.

Future study in this area should also consider the programmers' background (see section 6.5). This is difficult and may be easier to perform in the context of a proprietary setting, as was done by Ko et al (2007) and Sillito et al (2008). This approach (observation in a proprietary context) would complement the study reported on in this thesis, but would not directly inform on OS programmers.

6.7 Conclusion

This thesis began by identifying the need to explore information seeking behaviours of OS programmers. Observations were performed to characterize the sought information, with the main contributions consisting of the design of an Information Seeking Schema, identifying the most sought information types and determining how likely programmers are to gain responses for their information requests on mailing lists. The findings of this thesis largely reflect the academic articles in this area, documenting programmer's focus on implementation specific detail and team awareness (Ko et al., 2007, Gutwin et al., 2004). However interesting additional insights were gained with respect to the importance of the technology around software evolution in an OS context, and the programmers' design and documentation information needs. Also, a surprisingly low rate of response from the OS community to mailing list requests was noted.

Bibliography

- ALTHEIDE, D. 1987. *Ethnographic Content Analysis*, Human Science Press.
- ANGIONI, M., SANNA, R. & SOR, A. Defining a Distributed Agile Methodology for an Open Source Scenario. First International Conference on Open Source Systems, 11-15th July 2005 2005 Genova.
- ANIK, Z. & BAYKOÇ, Ö. F. 2011. Comparison of the most popular object-oriented software languages and criteria for introductory programming courses with analytic network process: A pilot study. *Computer Applications in Engineering Education*, 19, 89-96.
- BELADY, L. A. & LEHMAN, M. M. 1976. A model of large program development. *IBM Systems Journal*, 15, 225-252.
- BITSCH, V. 2005. Qualitative Research: A Grounded Theory Example and Evaluation Criteria. *Journal of Agribusiness*, 23, 75-91.
- BLACK, B. & RABINS, P. 2006. Qualitative Research in Psychogeriatrics *Guest Editorial - International Psychogeriatrics* ,*International Psychogeriatric Association*.
- BOGDAN, R. C. & BIKLEN, S. K. 1998. *Qualitative Research in Education. An Introduction to Theory and Methods.*, 160 Gould St., Needham Heights, MA 02194, Allyn & Bacon, A Viacom Company.
- BONACCORSI, A. & ROSSI, C. 2003. Why Open Source software can succeed. *Research Policy*, 32, 1243-1258.
- BRADAC, M. G., PERRY, D. E. & VOTTA, L. G. 1994. Prototyping a process monitoring experiment. *Software Engineering, IEEE Transactions on*, 20, 774-784.
- BRETTTHAUER, D. W. 2002. Open source software: a history. *Information Technology and Libraries*, 21, 3-10.
- BROOKS, R. 1977. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9, 737-751.
- BROOKS, R. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18, 543-554.
- BSF. 2007. *Mailing List for Java Bean Scripting Framework (BSF) Developers* [Online]. Available: <http://jakarta.apache.org/site/mail2.html> [Accessed November 2007].

- BUCKLEY, J. 1994. *DBSUM : diagram based software understanding and maintenance*. M. Sc., University of Limerick.
- BUCKLEY, J. 2002. *System Monitoring: A Tool for Capturing Software Engineers' Information-Seeking Behaviour*. PhD University of Limerick.
- BUCKLEY, J. 2009. Requirements-Based Visualization Tools for Software Maintenance and Evolution. *Computer*, 42, 106-108.
- BUCKLEY, J. 2010. *RE: The Difference between Concept and Feature Location*.
- BUCKLEY, J., O'BRIEN, M. P. & POWER, N. Empirically Refining a Model of Programmers' Information Seeking Behaviour During Software Maintenance. 18th Annual Psychology of Programming Interest Group (PPIG) Workshop,, 2006 Brighton, UK.
- BUDGE, I. & KLINGEMANN, H. D. 2001. *Mapping Policy Preferences: Estimates for Parties, Electors and Governments 1945-1998* Oxford University Press.
- CASE, D. O. 2007. *Looking for Information : A Survey of Research on Information Seeking, Needs and Behavior*, Academic Press.
- CHARMAZ, K. 2006. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*, SAGE.
- CHEN, K. & RAJLICH, V. Case Study of Feature Location Using Dependence Graph. *In: CLAV, R., ed., 2000. 241-241*.
- CHEN, X. & DAGNAT, F. 2011. Dynamic Reconfiguration on Java Internship bibliography. Rennes, France: Institut de recherche en informatique et systèmes aléatoires (IRISA).
- CLEARY, B. 2007. *Assisting Concept Location in Software Comprehension*. Ph. D in Computer Science, University of Limerick.
- CLEARY, B. & EXTON, C. Assisting Concept Location in Software Comprehension. The 19th Annual Psychology of Programming Interest Group Conference (PPIG '07). 2nd - 6th July 2007 2007 Finland.
- CLEARY, B., EXTON, C., BUCKLEY, J. & ENGLISH, M. 2008. An empirical analysis of information retrieval based concept location techniques in software comprehension *Empirical Software Engineering*, 14, 93-130.
- CORRITORE, C. & WIEDENBECK, S. 2000. Direction and Scope of Comprehension-Related Activities by Procedural and Object Oriented

- Programmers: An Empirical Study. *8th International Workshop on Program Comprehension*.
- CORRITORE, C. L. & WIEDENBACK, S. 1991. What Do Novices Learn During Program Comprehension? *International Journal of Human-Computer Interaction*, 3.
- CURTIS, B., HERB KRASNER, AND NEIL ISCOE. 1988. A field study of the software design process for large systems. *Communications of the ACM*, 31, 1268-1287.
- DANIEL, S., STEWART, K. & DARCY, D. 2009a. Patterns of Evolution in Open Source Projects: A Categorization Schema and Implications. *Patterns of Evolution in Open Source Projects: A Categorization Schema and Implications*. Minnesota: Management Information Systems Research Center, Carlson School of Management, University of Minnesota.
- DANIEL, S., STEWART, K. & DARCY, D. 2009b. Patterns of Evolution in Open Source Projects: A Categorization Scheme and Implications. University of Pittsburgh.
- DARCY, D. 2009. *RE: Example for OS project categories*.
- DAVID M. BERRY 2004. Internet research: privacy, ethics and alienation: an open source approach. *Internet Research Issue: 4*, 14, 323 - 332.
- DE LUCIA, A., FASOLINO, A. R. & MUNRO, M. Understanding function behaviors through program slicing. International Workshop on Program Comprehension (IWPC'96), 1996. IEEE, 9-19.
- DEMPSEY, B. J., WEISS, D., JONES, P. & GREENBERG, J. 2002. Who is an open source software developer? *Commun. ACM*, 45, 67-72.
- DENNIS HOWIT, D. C. 2008. *Introduction to Research Methods in Psychology*, Essex, England, Pearson Education Limited.
- DENZIN, N. & LINCOLN, Y. 2000. *Handbook of Qualitative Research* Sage Productions Ltd.
- DÉTIENNE, F. 2002. *Software Design - Cognitive Aspects*, Springer.
- DEWEY, J. 1933. *How we think*, Boston, DC Heath.
- DOTZLER, A. 2009. *Firefox at 270 million users* [Online]. Available: http://weblogs.mozillazine.org/asa/archives/2009/05/firefox_at_270.html [Accessed 1 Feb 2010].

- DURES, E., RUMSEY, N., MORRIS, M. & GLEESON, K. 2011. Mixed Methods in Health Psychology. *Journal of Health Psychology*, 16, 332-341.
- EL-EMAM, K. 2001. Ethics and Open Source. *Empirical Software Engineering*, 6, 291-292.
- ELLIS, D. & HAUGAN, M. 1997. Modeling the Information Seeking Patterns of Engineers & Research Scientists in an Industrial Environment. *Journal of Documentation*, 53, 384-403.
- ELTINGE, E. & ROBERTS, C. 1993. Linguistic Content Analysis: A Method to Measure Science as Inquiry in Textbooks. *Journal of Research in Science Teaching*, Vol. 30, 65-83.
- FELLER, J. & FITZGERALD, B. 2002. *Understanding Open Source Software Development*, Addison-Wesley, Pearson Education Limited.
- FERNANDA, B. V., GAS & JUDITH, S. D. 1999. Chat circles. *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*. Pittsburgh, Pennsylvania, United States: ACM.
- FIELDING, R. T. 1999. Shared leadership in the Apache project. *Commun. ACM*, 42, 42-43.
- FINKELSTEIN, A. & KRAMER, J. 2000. Software Engineering: A Roadmap. *Conference on the Future of Software Engineering*. Limerick.
- FITZGERALD, B. 2004. A critical look at open source. *Computer*, 37, 92-94.
- FOWLER, M. 2003. Patterns [software patterns]. *Software, IEEE*, 20, 56-57.
- GACEK, C. & ARIEF, B. 2004. The many meanings of open source. *Software, IEEE*, 21, 34-40.
- GALLAGHER, K. B. & LYLE, J. R. 1991. Using program slicing in software maintenance. *Software Engineering, IEEE Transactions on*, 17, 751-761.
- GAMMA, E., HELM, R., JOHNSON, R. & VLISSIDES, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* USA, Addison-Wesley.
- GASPERSON, T. 2006. *To Iraq and back: Soldier uses Linux in war* [Online]. Linux. Available: <http://www.linux.com/archive/articles/56216> [Accessed 30 September 2009].
- GIBBS, G. R. 2009. *Analysing Qualitative Data*, London SAGE Publications Limited.

- GILMORE, D. J. & GREEN, T. R. G. 1984. Comprehension and Recall of Miniature Programs. *International Journal of Man-Machine Studies*, 21, 31-48.
- GLASER, B. & STRAUSS, A. 1968. *The discovery of grounded theory: strategies for qualitative research*, London, Weidenfeld and Nicolson.
- GOLAFSHANI, N. 2003. Understanding Reliability and Validity in Qualitative Research. *The Qualitative Report*, 8.
- GOOD, J. 1999. *Programming Paradigms, Information Types and Graphical Representations : Empirical Investigations of Novice Program Comprehension*. PhD Thesis, The University of Edinburgh.
- GROUP, S. 2004. SAHANA : Home of the Free and Open Source Disaster Management System [Online]. Sahana Group. Available: <http://www.sahana.lk/> [Accessed 30 September 2009].
- GUTWIN, C., PENNER, R. & SCHNEIDER, K. 2004. Group awareness in distributed software development. *Proceedings of the 2004 ACM conference on Computer supported cooperative work*. Chicago, Illinois, USA: ACM.
- HANCOCK, B. 1998. Trent Focus for Research and Development in Primary Health Care: An Introduction to Qualitative Research. *Trent Focus*.
- HARRIS, N. & CILLIERS, C. A Program Beacon Recognition Tool. International Conference on Information Technology Based Higher Education and Training, 2006. ITHET '06. 7th 10-13 July 2006. 216-225.
- HARWOOD, I. 2002. *Developing Scenarios for Post-Merger and Acquisition Integration: A Grounded Theory of Risk Bartering*. Unpublished PhD, University of Southampton.
- HAYDEN, K. A. 2001. *Information Seeking Models* [Online]. Available: <http://people.ucalgary.ca/~ahayden/seeking.html> [Accessed 10 Feb 2010].
- HERTEL, G., NIEDNER, S. & HERRMANN, S. 2003. Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. *Research Policy*, 32, 1159-1177.
- HOEPFL, M. C. 1997. Choosing Qualitative Research: A Primer for Technology Education Researchers. *Journal of Technology Education*, 9.
- HOLLAND, R. 1999. Reflexivity. *Human Relations*, 52.

- HOLSTI, O. R. 1969. *Content Analysis for the Social Sciences and Humanities*, Reading, MA, Addison-Wesley.
- HOWITT, D. & CRAMER, D. 2008. *Introduction to Research Methods in Psychology*, Essex, England, Pearson Education Limited.
- IEEE 1991. *Standard Glossary of Software Engineering Terminology, IEEE 610.12-1990*, Institute of Electrical & Electronic Engineers Inc, New York,.
- IEEE 1998. *IEEE std. 1219-1998: Standard for software maintenance*, IEEE Press.
- INITIATIVE, O. S. 2009. *The Open Source Definition* [Online]. Available: <http://www.opensource.org/docs/osd> 2009].
- INTERNATIONAL, N. R. 2009. *Shadow 200 RQ-7 – Tactical Unmanned Aircraft System, USA* [Online]. Available: <http://www.army-technology.com/projects/shadow200uav/>.
- JAKARTA. 2007. *The Apache Jakarta Project* [Online]. Available: <http://jakarta.apache.org/>.
- JARVELIN, K. & REPO, A. On the impacts of modern information technology on information needs and seeking: A framework. *In: DIETSCHMANN, H. J., ed. Representation and exchange of knowledge as a basis of information processes*, 1983 Amsterdam: North-Holland., 207-230.
- JARVELIN, K. & REPO, A. A taxonomy of knowledge work support tools. *In: FLOOD, B., WITIAK, J. & HOGAN, T., eds. Challenges to an Information Society. Proc.47th ASIS Annual Meeting, Oct. 21-25, 1984. 1984 Philadelphia. Knowledge Industry Publications., 59-62.*
- JARVELIN, K. & WILSON, T. 2003. On conceptual models for information seeking and retrieval research. *Information Research*, 9.
- JDT. 2003. *Mailing List For Java Development Tool (JDT) Developers* [Online]. Available: <http://dev.eclipse.org/mhonarc/lists/jdtdev/maillist.html> [Accessed January 2008].
- JICK, T. D. 1979. Mixing Qualitative and Quantitative Methods: Triangulation in Action. *Administrative Science Quarterly*, 24, 602-611.
- JOHNSON, R. B. & ONWUEGBUZIE, A. J. 2004. Mixed Methods Research: A Research Paradigm Whose Time Has Come. *Educational Researcher*, 33, 14-26.

- JULIEN, H. 1996. A Content Analysis of the Recent Information Needs and Uses Literature. *Library & Information Science Research*, Vol. 18, 53-65.
- KARUS, S. & GALL, H. 2011. A Study of Language Usage Evolution in Open Source Software. 02/2011 ed.: ARXIV.
- KELLY, T. & BUCKLEY, J. Cognitive levels and Software Maintenance Sub-tasks. *In: EXTON, C., ed. 21st Working Conference on The Psychology of Programmers Interest Group, 2009 University Of Limerick , Ireland.*
- KELSEY, K. 2003. *Grounded Theory Designs*. [Online]. Available: Powerpoint presentation found at <http://www.okstate.edu/ag/agedcm4h/academic/aged5980/power/598314.ppt>
- KEMERER, C. F. & SLAUGHTER, S. 1999. An empirical approach to studying software evolution. *Software Engineering, IEEE Transactions on*, 25, 493-509.
- KERSTEN, M. & MURPHY, G. C. 2005. Mylar: a degree-of-interest model for IDEs. *Proceedings of the 4th international conference on Aspect-oriented software development*. Chicago, Illinois: ACM.
- KINGREY, K. P. 2002. Concepts of Information Seeking and Their Presence in the Practical Library Literature. *Library Philosophy & Practice*, 4.
- KO, A. J., DELINE, R. & VENOLIA, G. Information Needs in Collocated Software Development Teams. 29th International Conference on Software Engineering (ICSE'07), 2007.
- KOENEMANN, J. & ROBERTSON, S. P. Expert Problem Solving Strategies for Program Comprehension. SIGCHI Conference on Human Factors in Computing, 1991a. ACM Press.
- KOENEMANN, J. & ROBERTSON, S. P. 1991b. Expert problem solving strategies for program comprehension. *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*. New Orleans, Louisiana, United States: ACM.
- KOPONEN, T. & HOTTI, V. 2005. Open source software maintenance process framework. *Proceedings of the fifth workshop on Open source software engineering*. St. Louis, Missouri: ACM.
- KRIKELAS, J. 1983. Information-seeking Behaviour: Patterns and Concepts *Drexel Library Quarterly*, 19, 5-20.

- KRIPPENDORFF, K. 1980. *Content Analysis: An Introduction to Its Methodology*, Newbury Park, CA, Sage Publications.
- KRIPPENDORFF, K. 2004. *Content analysis: An introduction to its methodology*, Sage Publications.
- KUHLTHAU, C. C. 1988. Developing a Model of the Library Search Process: Investigation of Cognitive and Affective Aspects. *Reference Quarterly*, 28, 232-242.
- KUHLTHAU, C. C. 1993. *Seeking Meaning: A Process Approach to Library and Information Services*, New York, Greenwood Publishing.
- LAKHOTIA, A. 1994. Understanding someone else's code: Analysis of experiences. *Journal of Systems and Software*.
- LETHBRIDGE, T. C., SINGER, J. & FORWARD, A. 2003. How software engineers use documentation: the state of the practice. *Software, IEEE*, 20, 35-39.
- LETOVSKY, S. 1986. Cognitive Process in Program Comprehension. *In: SOLOWAY, E., ed. First Workshop on Empirical Studies of Programmers*, 5-6 June 1986 Washington, DC. Ablex Publishing Corporation, 58-79.
- LETOVSKY, S. 1987. Cognitive Processes In Program Comprehension. *Journal of Systems and Software*, 7, 325-339.
- LIENTZ, B. P., SWANSON, E. B. & TOMPKINS, G. E. 1978. Characteristics of application software maintenance. *Commun. ACM*, 21, 466-471.
- LITTMAN, D. C., PINTO, J., LETOVSKY, S. & SOLOWAY, E. Mental models and software maintenance. *Empirical Studies of Programmers: 1st Workshop*, 1986. 80-98.
- MARCHIONINI, G. 1997. *Information seeking in electronic environments*, New York, Cambridge University Press.
- MARCUS, A., RAJLICH, V., BUCHTA, J., PETRENKO, M. & SERGEYEV, A. Static Techniques for Concept Location in Object-Oriented Code. 13th International Workshop on Program Comprehension (IWPC'05), 2005.
- MARSHALL, G. 1998. *A Dictionary of Sociology*, Oxford University Press.
- MAYRHAUSER, A. V. & VANS, A. M. From code understanding needs to reverse engineering tool capabilities. Sixth International Conference on Computer-Aided Software Engineering (CASE'93), 1993. 230-239.

- MAYRHAUSER, A. V. & VANS, A. M. 1995. Program Understanding: Models and Experiments. *Advances in Computers*, 40, 25-26.
- MOCKUS, A., FIELDING, R. T. & HERBSLEB, J. 2000. A case study of open source software development: the Apache server. *Proceedings of the 22nd international conference on Software engineering*. Limerick, Ireland: ACM.
- MOCKUS, A., FIELDING, R. T. & HERBSLEB., J. D. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11, 309-346.
- MOORE, L. & SAVAGE, J. 2002. Participant observation, informed consent and ethical approval. *Nurse Researcher*, 9, 58-69.
- MULLER, H. A., WONG, K. & TILLEY, S. R. Understanding software systems using reverse engineering technology. The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS 1994), 1994.
- NASSERI, E. 2009. *An Empirical Investigation of Inheritance Trends in Java OSS Evolution*. Brunel University United Kingdom.
- NEILL, J. 2007. *Qualitative versus Quantitative Research: Key Points in a Classic Debate* [Online]. Available: <http://wilderdom.com/research/QualitativeVersusQuantitativeResearch.html> [Accessed 19 March 2010].
- NETCRAFT. 2009a. *Most Reliable Hosting Company Sites in October 2009* [Online]. Available: http://news.netcraft.com/archives/2009/11/03/most_reliable_hosting_company_sites_in_october_2009.html.
- NETCRAFT. 2009b. *Web Server Survey* [Online]. Available: <http://news.netcraft.com/> [Accessed 1st Dec 2009].
- NETCRAFT. 2009c. *White House goes Open Source* [Online]. Available: http://news.netcraft.com/archives/2009/10/27/white_house_goes_open_source.html.
- NETCRAFT. 2010. *Search Web by Domain : Results for .microsoft.com* [Online]. Available: <http://searchdns.netcraft.com/?restriction=site+contains&host=.microsoft.com&lookup=wait..&position=limited>.
- NIEDŹWIEDZKA, B. 2003. A proposed general model of information behaviour. *Information Research*, 9.

- NIELS, J. 2001. Putting it all in the trunk: incremental software development in the FreeBSD open source project. *Information Systems Journal*, 11, 321-336.
- O'BRIEN, M. P. 2007. *Evolving a Model of the Information-Seeking Behaviour of Industrial Programmers*. Doctor of Philosophy in Computer Science, University of Limerick.
- O'BRIEN, M. P., BUCKLEY, J. & SHAFT, T. M. 2004. Expectation-based, inference-based, and bottom-up software comprehension. *Journal of Software Maintenance and Evolution: Research and Practice*, 16.
- O'BRIEN, M. P. & JIM BUCKLEY. Modelling the Information-Seeking Behaviour of Programmers - An Empirical Approach. 13th International Workshop on Program Comprehension (IWPC'05), 2005. 125-134.
- O'BRIEN, M. P., SHAFT, T. M. & BUCKLEY, J. An Open-Source Analysis Schema for Identifying Software Comprehension Processes. 13th Workshop of the Psychology of Programming Interest Group, April 2001 2001 Bournemouth UK.
- O'SHEA, P. A. 2006. *An Investigation of Views and Abstractions Employed by Software Engineers during Software Maintenance - An Empirically Founded set of Guidelines for Visualisation Tools Supporting Comprehension*. PhD Thesis.
- O'SHEA, P. A. 2007. *RE: 'Location' Information Type*.
- O'SHEA, P. A. & EXTON, C. The Application of Content Analysis to Programmer Mailing Lists as a Requirements Method for a Software Visualisation Tool. International Workshop on Software Technology and Engineering Practice (STEP'04), 2004. IEEE, 30-39.
- O'BRIEN, M. P. 2007. *Evolving a Model of the Information-Seeking Behaviour of Industrial Programmers*. Doctor of Philosophy in Computer Science, University of Limerick.
- OATES, B. J. 2006. *Researching Information Systems and Computing*, Sage Publications Ltd.
- PANDIT, N. R. 1996. The Creation of Theory:A Recent Application of the Grounded Theory Method. *The Qualitative Report* [Online], 2. Available: <http://www.nova.edu/ssss/QR/QR2-4/pandit.html> [Accessed 14th April 2008].

- PENNINGTON, N. Comprehension strategies in Programming. Empirical studies of programmers: second workshop, 1987a. Ablex Publishing Corp., 100-113.
- PENNINGTON, N. 1987b. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19, 295-341.
- PERRY, D., PORTER, A. & VOTTA, L. 1997. A Primer on Empirical Studies *International Conference on Software Maintenance*.
- PETERSON, C., BETTES, B. & SELIGMAN, M. 1985. Depressive Symptoms and Unprompted Causal Attributions: Content Analysis *Behaviour Research and Therapy* Vol. 23, 379-82.
- POSHYVANYK, D. & MARCUS, A. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on, 26-29 June 2007 2007. 37-48.
- POSHYVANYK, D., MARCUS, A. & RAJLICH, V. C. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 33, 420-431.
- POWER, N. 2002. *A Grounded Theory of Requirements Documentation in the Practice of Software Development*. PhD, Dublin City University.
- POWER, N. 2009. *RE: Saturation in Grounded Theory*. Type to SHARIF, K. Y. & BUCKLEY, J.
- PRECHELT, L., UNGER, B., PHILIPPSSEN, M. & TICHY, W. 1998. Re-evaluating inheritance depth on the maintainability of object-oriented software. *International Journal of Empirical Software Engineering*, 1-16.
- PRESSMAN, R. S. 2000. *Software Engineering: A Practitioner's Approach*, Shoppenhangers Road, Maidenhead, Berkshire SL6 2QL, England., McGraw-Hill Publishing Company.
- RAJLICH, V. & WILDE, N. The role of concepts in program comprehension. Program Comprehension, 2002. Proceedings. 10th International Workshop on, 2002. 271-278.
- RAJLICH, V. C. 2004. Incremental Change in Object-Oriented Programming. *In*: PRASHANT, G. (ed.).

- RAYMOND, E. S. 2001a. *The Cathedral and the Bazaar. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* O'Reilly Associates, Inc.
- RAYMOND, E. S. 2001b. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly & Associates, Inc.
- RESIPROCAT. 2009. *reSIProcate project* [Online]. Available: <http://list.resiprocate.org/archive/resiprocate-devel/mail3.html> [Accessed Jan 2010].
- RIESSMAN, C. K. 1994. *Narrative analysis*, SAGE.
- RIGBY, P. C. & STOREY, M. A. Understanding broadcast based peer review on open source software projects. *Software Engineering (ICSE), 2011 33rd International Conference on, 21-28 May 2011* 2011. 541-550.
- ROSENBERG, J., SCHULZRINNE, H., CAMARILLO, G., JOHNSTON, A., PETERSON, J., SPARKS, R., HANDLE, M. & SCHOOLER, E. 2002. *SIP: Session Initiation Protocol* [Online]. Available: <ftp://ftp.rfc-editor.org/in-notes/rfc3261.txt> [Accessed 23 Feb 2010].
- SCHMIDT, D. C. & PORTER, A. 2001. Leveraging open-source communities to improve the quality performance of open-source software. In: *Making Sense of the Bazaar*. Feller, J., Fitzgerald, B. and van der Hoek, A (eds). *First Workshop on Open Source Software 23rd ICSE Conference*. Toronto.
- SEAMAN, C. B. The Information Gathering Strategies of Software Maintainers. *International Conference on Software Maintenance (ICSM02), 2002*. IEEE, 141-149.
- SHAFT, T. M. 2001. *RE: Coders Manual*. Type to CORRESPONDENCE, P.
- SHAFT, T. M. & VESSEY, I. 1995. The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research*, 6, 286-299.
- SHARIF, K. Y. & BUCKLEY, J. Developing Schema for Open Source Programmers' Information-Seeking. *International Symposium on Information Technology 2008 (ITSIM '08), September 2008* 2008a Kuala Lumpur , Malaysia. Kuala Lumpur: IEEE.
- SHARIF, K. Y. & BUCKLEY, J. Observing Open Source Programmers' Information Seeking. *The 20th Annual Psychology of Programming Interest Group Conference, 10th - 12th September 2008* 2008b Lancaster University, Lancaster, United Kingdom.

- SHARIF, K. Y. & BUCKLEY, J. Further Observation of Open Source Programmers' Information Seeking. *In*: EXTON, C. & BUCKLEY, J., eds. Psychology of Programming Interest Group, 2009a Limerick, Ireland. PPIG.
- SHARIF, K. Y. & BUCKLEY, J. Observation of Open Source programmers' information seeking. Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on, 2009b Vancouver, British Columbia, Canada. IEEE Computer Society, 307-308.
- SHEN, W., HAO, Q. & LI, W. 2008. Computer supported collaborative design: Retrospective and perspective. *Computers in Industry*, 59, 855-862.
- SHNEIDERMAN, B. & MAYER, R. 1979. Syntactic / semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, 8, 219-238.
- SILLITO, J., MURPHY, G. C. & DE VOLDER, K. 2008. Asking and Answering Questions during a Programming Change Task. *Software Engineering, IEEE Transactions on*, 34, 434-451.
- SILLITO, J., MURPHY, G. C. & VOLDER, K. D. 2006. Questions programmers ask during software evolution tasks. *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. Portland, Oregon, USA: ACM.
- SIM, S. E. 1998. *Supporting Multiple Program Comprehension Strategies During Software Maintenance*. Masters Thesis, University of Toronto.
- SINGER, J. Work Practices of Software Maintenance Engineers. International Conference on Software Maintenance (ICSM '98), 1998 Washington, Federal District of Columbia, USA. 139-145.
- SINGER, J. & LETHBRIDGE, T. Studying work practices to assist tool design in software engineering. 6th International Workshop on Program Comprehension (IWPC'98), 1998. IEEE, 173-179.
- SINGER, J., LETHBRIDGE, T., VINSON, N. & ANQUETIL, N. An Examination of Software Engineering Work Practices. Centre for Advanced Studies on Collaborative research, 1997 Toronto, Canada.
- SMITH, C. P. 2000. Chapter 12 : Content Analysis and Narrative Analysis *In*: REIS, H. T. & JUDD, C. M. (eds.) *Handbook of research methods in social and personality psychology*. Cambridge University Press.

- SNYDER, I. 1995. Multiple Perspectives in Literacy Research: Integrating the Quantitative and Qualitative. *Language and Education*, 9, 45-59.
- SOLOWAY, E. & EHRLICH, K. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10, 595-609.
- SOLOWAY, E. & EHRLICH, K. 1986. Empirical studies of programming knowledge. *Readings in artificial intelligence and software engineering*. Morgan Kaufmann Publishers Inc.
- SOLOWAY, E., LAMPERT, R., LETOVSKY, S., LITTMAN, D. & PINTO, J. 1988. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31, 1259-1267.
- SOMMERVILLE, I. 2004. *Software Engineering*, Addison-Wesley.
- SOURCEFORGE. 2001. *Email Archive: eboard-devel (read-only)* [Online]. Available: http://sourceforge.net/mailarchive/forum.php?thread_name=Pine.LNX.4.30.0104290400270.19983-100000%40moriamit.edu&forum_name=eboard-devel [Accessed Jan 2010].
- SOURCEFORGE. 2004. *SwingWT* [Online]. Available: http://sourceforge.net/mailarchive/forum.php?forum_name=swingwt-developers&max_rows=25&style=threaded&viewmonth=200401 [Accessed Jan 2010].
- SOURCEFORGE. 2007. *Browse Files for SwingWT* [Online]. Available: <http://sourceforge.net/projects/swingwt/files/> [Accessed 23 February 2010].
- SOUSA, CASTRO, M. J. & MOREIRA., H. M. A Survey on the Software Maintenance Process. International Conference on Software Maintenance, November 1998 1998 Bethesda, MD. 265-274.
- SOUZA, C. D., FROEHLICH, J. & DOURISH, P. 2005. Seeking the source: software source code as a social and technical artifact. *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*. Sanibel Island, Florida, USA: ACM.
- SPINDLER, G. 1982. *Doing the Ethnography of Schooling*, New York, CBS Publishing.
- STOREY, M.-A. 2005. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. *Proceedings of the 13th International Workshop on Program Comprehension*. IEEE Computer Society.

- STOREY, M.-A. 2006. Theories, tools and research methods in program comprehension: <i>past, present and future&/i>. *Software Quality Journal*, 14, 187-208.
- STRAUSS, A. & CORBIN, J. 1998. *Basics of qualitative research: Techniques and procedures for developing grounded theory (2nd ed.)*, Thousand Oaks, CA, US: Sage Publications, Inc.
- STRAUSS, A. L. 1987. *Qualitative Analysis for Social Scientist*, New York, Cambridge University Press.
- SWANSON, E. B. The dimension of maintenance. 2nd International Conference on Software Engineering (ICSE'76), 1976. 492–497.
- TANNEN, D. 2001. *Handbook of Discourse Analysis*, Oxford and Cambridge, MA, Basil Blackwell.
- THOMAS, D. L. & BRAD, A. M. 2010. On the importance of understanding the strategies that developers use. *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*. Cape Town, South Africa: ACM.
- THOMAS, R. G. G. 2000. Instructions and descriptions: some cognitive aspects of programming and similar activities. *Proceedings of the working conference on Advanced visual interfaces*. Palermo, Italy: ACM.
- TOOMERE, T. Personal Success Stories in the Estonian Press. The Fourth Nordic Conference on the Anthropology of Post-Socialism, 2002 Tallinn Pedagogical University.
- TORVALDS, L. & DIAMOND, D. 2001. *Just for fun : The Story of An Accidental Revolutionary*, New York, HarperCollins.
- VON MAYRHAUSER, A. & VANS, A. M. 1995. Program comprehension during software maintenance and evolution. *Computer*, 28, 44-55.
- VON MAYRHAUSER, A. V., A. M. From code understanding needs to reverse engineering tool capabilities. Sixth International Conference on Computer-Aided Software Engineering (CASE'93), 1993. 230-239.
- WESTBROOK, L. 1994. Qualitative research methods: A review of major stages, data analysis techniques, and quality controls. *Library & Information Science Research*, 16, 241-254.

- WIKIPEDIA. 2008. *Session Initiation Protocol* [Online]. Available: http://en.wikipedia.org/wiki/Session_Initiation_Protocol [Accessed 23 February 2010].
- WIKIPEDIA. 2010. *Java version history* [Online]. Available: http://en.wikipedia.org/wiki/Java_version_history [Accessed 28 Dec 2010 2010].
- WILDE, N., GOMEZ, J. A., GUST, T. & STRASBURG, D. Locating user functionality in old code. *Software Maintenance, 1992. Proceedings., Conference on, 9-12 Nov 1992* 1992. 200-205.
- WILSON, C. 2007. Network Centric Operations: Background and Oversight Issues for Congress," Congressional Research Service Report for Congress. *CRS Report for Congress*. United States Department of Defense.
- WILSON, T. D. 1981. On User Studies and Information Needs *Journal of Documentation*, 37, 3 -15.
- WILSON, T. D. 1997. Information behaviour: An interdisciplinary perspective. *Information Processing & Management*, 33, 551-572.
- WILSON, T. D. 1999. Models of Information Behaviour Research. *Journal of Documentation*, 55, 249-270.
- WU, C.-G., GERLACH, J. H. & YOUNG, C. E. 2007. An empirical analysis of open source software developers' motivations and continuance intentions. *Information & Management*, 44, 253-262.
- YANG, S. C. 1997. Information seeking as problem-solving using a qualitative approach to uncover the novice learners' information-seeking processes in a perseus hypertext system. *Library & Information Science Research*, 19, 71-94.
- ZAYOUR, I. & LETHBRIDGE, T. C. Adoption of reverse engineering tools: a cognitive perspective and methodology. 9th International Workshop on Program Comprehension (IWPC'01), 2001. IEEE, 245-258.

Appendix A

You replied on 4/6/2009 8:13 PM.

KhaironiYatim.Sharif

From: David Darcy [David.Darcy@im.ue] **Sent:** Fri 2/27/2009 5:20 PM
To: KhaironiYatim.Sharif
Cc:
Subject: RE: Example for OS project categories.
Attachments:

Khaironi,

I'll keep you posted with regard to the paper being published, fingers crossed.

Here are some example projects in the categories you mentioned:

User centered

nmsti

Eboard

Controlled

Pocketwarrior

Klapjack

Atypical

Reciprocate

Zige

I hope these help,

Dave.

-----Original Message-----

From: KhaironiYatim.Sharif [mailto:khaironiyatim.sharif@ul.ie]
Sent: 16 February 2009 16:37

<https://webmail.ul.ie/exchange/KhaironiYatim.Sharif/Inbox/Research/RE:%20Example%20...> 8/3/2011

To: David Darcy
Subject: Example for OS project categories.

Dear David,

Thanks you very much for forwarding me your draft paper. I found it really interesting and think that it will have a really high citation index (as all other papers in the OS area that claim to be well-read will have to cite yours for contextualization). In fact, I'd be really interested in finding out where and when this is eventually published so that I can cite it myself.

One thing I found surprising was that you didn't give an example of a project in each category. As I'm doing a grounded-theory study of OS mailing lists (the questions developers ask during their maintenance), I would like to review at least one example for each of your first 3 categories (user-centred, control and atypical), as our study focuses on maintenance and the other 3 categories seem to have short maintenance life-spans.

Is there any possibility that you could point me in the direction of 2-3 examples of each of the first 3 categories?

Many thanks for any help you can provide on this, and again, many thanks for a really interesting read!

Regards,

Khaironi.

This email and any files transmitted with it are
confidential and intended solely for the use of the
individual or entity to whom they are addressed. If
you have received this email in error please notify
the system manager at postmaster@iml.uia.

<https://webmail.ul.ie/exchange/KhaironiYatim.Sharif/Inbox/Research/RE:%20Example%20...> 8/3/2011

Figure A1 – Personal Email Communication With One Author Of Daniel et al (2009).

Table A1 - Information Focus Updated During 2nd Observation.

Info. Focus	Definition and Example
Integrated Development Environment (IDE)	<p>Question asking about the IDE used in the project. These could be differentiated into 2 main aspects when combined with <i>Information Aspect</i> categories :</p> <ul style="list-style-type: none"> • IDE – Guide. Question asking for detailed guidance on using the IDE component to achieve a goal. Normally these are asked in <i>How</i> questions. For example: <i>“How to use Eclipse (for java) with CVS.”</i> • IDE - Feature. Question that ask about features of the IDE component. Normally these are asked as <i>What</i> questions such as: <i>“I was wondering if Eclipse, has the capability for enabling the development of the EJBs and deploying it in the container for the any application sever that might come with it.”</i> This would also asked in <i>Where</i> questions such as <i>“where can i configure quickDiff in eclipse 3.0m7?”</i>
Communication Channel	<p>Question that refer to the communication channel such as mailing lists, or bug reporting channels. These could again be differentiated into 2 main aspects:</p> <ul style="list-style-type: none"> • <i>Communication Channel</i> – Guide. Question that ask for detailed guidance on using the channel. Normally asked in <i>How</i> questions. For example : <i>Is there something that needs to be done to get the SVN commits i do have notifications sent to the BSF dev list?</i> • <i>Communication Channel-Choice</i>. Questions reflects a choice between one or another channel. Normally asked in <i>Which</i> questions. For example: <i>“can we use JIRA for bug reporting for this project instead .. Thoughts ?”</i>
Operation Environment	<p>Question asking about the operation environment of the IDE, server and any related surrounding technical context that is involved in the application running such as operating system, and plug-ins. Likewise, this can be differentiated into guidance and choice requests:</p> <ul style="list-style-type: none"> • <i>Operation Environment</i> – Guide. Question that ask for detailed guidance on using specific components for successful execution of the application. For example: <i>“Can I invoke update manager using bash scripts to install our plugins and features?..”</i> • <i>Operation Environment</i> - Choice. Question that reflects a choice between one or another operation environment. Normally asked in <i>Which</i> questions. For example : <i>“Is Apache easier to deal with than Jakarta?”</i>
Support Required	<p>Questions that ask another community member to take on responsibility or tasks. Example: <i>“There are 2 non-filed open issues..... Are there any taker? ”</i></p>
System Enhancement	<p>Questions that aim to understand the code in order to make evolutionary change. Example : <i>“...but I need to understand the refactoring currently in Eclipse now. Can anyone suggest me where about in the code is a good starting point in</i></p>

	<i>understanding how the component work"</i>
System Bug	Questions that aim to understand the code in order to trace a bug. Example : <i>"(Given a situation..)I have no idea why this is happening. Please help me solve this problem"</i>
Task-Test	Question related to testing. Example : <i>"Is renameParticipants working? I am doing a spike test but cannot get it work so far."</i>
Task-Implementation	Question about tasks that are related to Implementation. Note that this is not about comprehending the code but more directed at the task to be undertaken. Example : <i>"Maybe you need to post more code, or maybe you need to update ecs-1.4.1?"</i>
Documentation	Questions referring to the documentation: Example: <i>"Is there any Apache official guidelines on this?"</i>
System Design	Question referring to the system's design. Example : <i>"Is jdt.core.jdom built on top of jdt.core.dom? Can you get to the underlying jdom model?"</i> <i>"Is there any plan to add pattern functionality to the JDT UI, i.e.. (given an example)"</i>
File Configuration	Question about configuration management. Example : <i>" What is the distribution directory in the src zip/tgz? "</i>
Person	Question about the person in-charge or responsible for some task. Example : <i>"Who is the team / person in charge for documentation?"</i>
Distribution / Release	Question about distribution or release of source code / software product to public. Example : <i>" What do people think about cutting a release of BSF V3? Having a real release instead of just a SNAPSHOT build makes it mush easier for other projects to start using BSF and getting users is the best way to get BSF tested and improved."</i>

Table A2 – Information Aspect Updated During 2nd Observation

Info. Aspect	Definition and Example
How	<p>Questions which attempted to identify ‘<i>how some goal of the system is achieved</i>’, ‘<i>how some software tool feature is employed</i>’ and also ‘<i>how to proceed</i>’. This could be differentiated into several sub-categories such as :</p> <ul style="list-style-type: none"> • <i>How – System (how some goal of the system is achieved (code based) : “How X can be 5?”</i> in instances like this, the <i>How-System</i> subcategory could be equated to Letovsky’s (1987)Top-Down comprehension. • <i>How – Software Tool(how some software tool feature is employed) : “How do I compile my Java source in Eclipse?”</i> • <i>How – Procedure (how to proceed): “I am a little unsure how best to make a patch, I would imagine that easiest way to this accepted would be to make patch against the Head of the org.eclipse.jdt.junit at eclipse.org right?”</i>
What	<p>Questions which ask to define / identify something such as what source code or software tool elements do. This would also differentiated into several sub-categories such as :</p> <ul style="list-style-type: none"> • <i>What – System : “What is the value of X at this line”</i>. When referring to source code, these questions represent bottom-up program comprehension (Letovsky 1986). • <i>What – Software Tool: “What is the features of GTK that can be used with Motif?”</i> • <i>What – Configuration: “What is the .rep file?”</i>
Where	<p>Asking about the location of software artifacts, tool etc. Example: “<i>Where I can find the sources for plug in so I can create a patch?</i>”</p>
Who	<p>Asking for the relevant persons to seek for information or to perform a task. Example: “<i>Can someone please point me to the information development team that wrote the used documentation?</i>”</p>
Why	<p>Asking for the purpose / explanation of a system behaviour or rationale of design.</p> <ul style="list-style-type: none"> • <i>Why – System: “I am getting an exception being thrown when trying to create new java class and I was wondering if anyone could shed any light on why?”</i> This also typically represents bottom-up program comprehension by programmers (Letovsky 1986). • <i>Why – Design: “Why it is called IPackageFragment and not IPackage? getPackageFragment()?”</i>

Which	Question that reflects a choice between one and another subject (information focus). Example <i>“can we use JIRA for bug reporting for this project instead .. Thoughts ?”</i>
Relationship	Relationship between 2 or more things. It differs from other questions in that it directs itself at relationships between entities rather than at entities themselves, thus suggesting ‘analysis level’ information (Kelly and Buckley, 2009). Example: <i>“What is the dependence between PackageFragementRoot and PackageFragment?”</i>
Validation	Asking permission to do something. This strategy is normally related with Legality / Protocol. It seeks permission to do something. Example: <i>“I was just looking to submit a JIRA patch for the existing BSF 3.0 prototype code but there doesn't seem to be a JIRA project for BSF. How about I set up BSF in the ASF JIRA system and we can use that for any BSF 3.0 work?”</i>
Awareness	Question to make sure that the asker or the audience has up to date information about current issues in the team. Example : <i>“Do we need to tell anyone in Apache we're doing this?”</i>
Legality / Protocol	Questions about the protocol to follow within the project. Example: <i>“Did you got the approval to contribute your work to BSF? ”</i>
When	Questions asking about timeline or time of occurrence. Example : <i>When is the next BSF release expected?</i>

Appendix B

Appendix B

Table B1 - Information Focus Updated During 3rd Observation.

Info. Focus	Definition and Example
Integrated Development Environment (IDE)	<p>Question asking about the IDE used in the project. These could be differentiated into 2 main aspects when combined with <i>Information Aspect</i> categories :</p> <ul style="list-style-type: none"> • IDE – Guide. Question asking for detailed guidance on using the IDE component to achieve a goal. Normally these are asked in <i>How</i> questions. For example: <i>“How to use Eclipse (for java) with CVS.”</i> • IDE - Feature. Question that ask about features of the IDE component. Normally these are asked as <i>What</i> questions such as: <i>“I was wondering if Eclipse, has the capability for enabling the development of the EJBs and deploying it in the container for the any application sever that might come with it.”</i> This would also asked in <i>Where</i> questions such as <i>“where can i configure quickDiff in eclipse 3.0m7?”</i>
Communication Channel	<p>Question that refer to the communication channel such as mailing lists, or bug reporting channels. These could again be differentiated into 2 main aspects:</p> <ul style="list-style-type: none"> • <i>Communication Channel</i> – Guide. Question that ask for detailed guidance on using the channel. Normally asked in <i>How</i> questions. For example : <i>Is there something that needs to be done to get the SVN commits i do have notifications sent to the BSF dev list?</i> • <i>Communication Channel</i>-Choice. Questions reflects a choice between one or another channel. Normally asked in <i>Which</i> questions. For example: <i>“can we use JIRA for bug reporting for this project instead .. Thoughts ?”</i>
Operation Environment	<p>Question asking about the operation environment of the IDE, server and any related surrounding technical context that is involved in the application running such as operating system, and plug-ins. Likewise, this can be differentiated into guidance and choice requests:</p> <ul style="list-style-type: none"> • <i>Operation Environment</i> – Guide. Question that ask for detailed guidance on using specific components for successful execution of the application. For example: <i>“Can I invoke update manager using bash scripts to install our plugins and features?..”</i> • <i>Operation Environment</i> - Choice. Question that reflects a choice between one or another operation environment. Normally asked in <i>Which</i> questions. For example : <i>“Is Apache easier to deal with than Jakarta?”</i>
Support Required	<p>Questions that ask another community member to take on responsibility or tasks. Example: <i>“There are 2 non-filed open issues..... Are there any taker? ”</i></p>

System Enhancement	Questions that aim to understand the code in order to make evolutionary change. Example : <i>"...but I need to understand the refactoring currently in Eclipse now. Can anyone suggest me where about in the code is a good starting point in understanding how the component works"</i>
System Bug	Questions that aim to understand the code in order to trace a bug. Example : <i>"(Given a situation..)I have no idea why this is happening. Please help me solve this problem"</i>
Task-Test	Question related to testing. Example : <i>"Is renameParticipants working? I am doing a spike test but cannot get it work so far."</i>
Task-Implementation	Question about tasks that are related to Implementation. Note that this is not about comprehending the code but more directed at the task to be undertaken. Example : <i>"Maybe you need to post more code, or maybe you need to update ecs-1.4.1?"</i>
Documentation	Questions referring to the documentation: Example: <i>"Is there any Apache official guidelines on this?"</i>
System Design	Question referring to the system's design. Example: <i>"Is the implementation of generics expected to trigger major changes to the internal data structures?"</i> <i>"Is jdt.core.jdom built on top of jdt.core.dom? Can you get to the underlying jdom model?"</i> <i>"Is there any plan to add pattern functionality to the JDT UI, i.e.. (given an example)"</i>
File Configuration	Question about configuration management. Example : <i>" What is the distribution directory in the src zip/tgz? "</i>
Distribution / Release	Question about distribution or release of source code / software product to public. Example : <i>" What do people think about cutting a release of BSF V3? Having a real release instead of just a SNAPSHOT build makes it mush easier for other projects to start using BSF and getting users is the best way to get BSF tested and improved."</i>
Stage	Question asking about completion or stage of certain activities or a project. Example: <i>"I've not seen any noise on this list since I joined. Is there any life in the BSF sub-project??")"</i>.

Table B2 – Information Aspect Updated During 3rd Observation

Info. Aspect	Definition and Example
How	<p>Questions which attempted to identify ‘<i>how some goal of the system is achieved</i>’, ‘<i>how some software tool feature is employed</i>’ and also ‘<i>how to proceed</i>’. This could be differentiated into several sub-categories such as :</p> <ul style="list-style-type: none"> • <i>How – System (how some goal of the system is achieved (code based) : “How X can be 5?”</i> in instances like this, the <i>How-System</i> subcategory could be equated to Letovsky’s (1987)Top-Down comprehension. • <i>How – Software Tool(how some software tool feature is employed) : “How do I compile my Java source in Eclipse?”</i> • <i>How – Procedure (how to proceed): “I am a little unsure how best to make a patch, I would imagine that easiest way to this accepted would be to make patch against the Head of the org.eclipse.jdt.junit at eclipse.org right?”</i>
What	<p>Questions which ask to define / identify something such as what source code or software tool elements do. This would also differentiated into several sub-categories such as :</p> <ul style="list-style-type: none"> • <i>What – System : “What is the value of X at this line”. When referring to source code, these questions represent bottom-up program comprehension (Letovsky 1986).</i> • <i>What – Software Tool: “What is the features of GTK that can be used with Motif?”</i> • <i>What – Configuration: “What is the .rep file?”</i>
Where	<p>Asking about the location of software artifacts, tool etc. Example: “<i>Where I can find the sources for plug in so I can create a patch?</i>”</p>
Who	<p>Asking for the relevant persons to seek for information or to perform a task. Example: “<i>Can someone please point me to the information development team that wrote the used documentation?</i>”</p>
Why	<p>Asking for the purpose / explanation of a system behaviour or rationale of design.</p> <ul style="list-style-type: none"> • <i>Why – System: “I am getting an exception being thrown when trying to create new java class and I was wondering if anyone could shed any light on why?” This also typically represents bottom-up program comprehension by programmers (Letovsky 1986).</i> • <i>Why – Design: “Why it is called IPackageFragment and not IPackage? getPackageFragment()?”</i>
Which	<p>Question that reflects a choice between one and another subject (information focus). Example “<i>can we use JIRA for bug reporting for this project instead .. Thoughts ?</i>”</p>

Validation	Asking permission to do something. This strategy is normally related with Legality / Protocol. It seeks permission to do something. Example: "I was just looking to submit a JIRA patch for the existing BSF 3.0 prototype code but there doesn't seem to be a JIRA project for BSF. How about I set up BSF in the ASF JIRA system and we can use that for any BSF 3.0 work?"
Awareness	Question to make sure that the asker or the audience has up to date information about current issues in the team. Example : "Do we need to tell anyone in Apache we're doing this?"
Legality / Protocol	Questions about the protocol to follow within the project. Example: "Did you got the approval to contribute your work to BSF? "
When	Questions asking about timeline or time of occurrence. Example : <i>When is the next BSF release expected?</i>