

Open Research Online

The Open University's repository of research publications and other research outputs

Production and use of documentation in scientific software development

Thesis

How to cite:

Pawlik, Aleksandra (2014). Production and use of documentation in scientific software development. PhD thesis The Open University.

For guidance on citations see [FAQs](#).

© 2014 Aleksandra Pawlik

Version: Version of Record

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's [data policy](#) on reuse of materials please consult the policies page.

oro.open.ac.uk

Production and use of documentation in scientific software development



Aleksandra Pawlik
Department of Computing
The Open University

A thesis submitted for the degree of
Doctor of Philosophy in Computing

2013

Abstract

Software is becoming ubiquitous in science. The success of the application of scientific software depends on effective communication about what the software does and how it operates. Documentation captures the communication about the software. For that reason, practices around scientific software documentation need to be better understood. This thesis presents four qualitative empirical studies that look in depth at the production and use of documentation of scientific software. Together, the studies provide evidence emphasising the importance of documentation and shows the handshake between written documentation and the informal, ephemeral information exchange that happens within the community.

Four reasons behind the obstacles to producing effective scientific software documentation are identified: 1) the insufficient resources; 2) lack of incentives for researchers; 3) the influence of the community of practice; 4) the necessity of keeping up with the regular advancements of science. Benefits of the process of producing documentation are also identified: 1) aiding reasoning; 2) supporting reproducibility of science; 3) in certain contexts, expanding the community of users and developers around the software. The latter is investigated through a case study of documentation crowdsourcing.

The research reveals that there is a spectrum of users, with differing needs with respect to documentation. This, in turn, requires different approaches in addressing their needs. The research shows that the view of what constitutes documentation must be broad, in order to recognise how wide a range of resources (e.g., formal documents, email, online fora, comments in the source code) is actually used in communicating knowledge about scientific software. Much of the information about the software resides within the community of practice (and may not be documented). These observations are of practical use for those producing documentation in different contexts of scientific software development, for example providing guidance about engaging a community in crowdsourcing documentation.

Acknowledgments

I would like to thank my supervisors Dr Judith Segal, Prof. Helen Sharp and Prof. Marian Petre for their ongoing support. Their feedback, patience and ability to motivate me allowed me to complete this thesis. I want to also thank the participants from all the studies as well as everyone who helped me in my data collection. Without all of them this research would never have happened. I am grateful to everybody who read and commented on my work - their input helped me to find the right path many times. I owe a big thank you to my friends and colleagues, especially those from the Open University, who have always been ready to share my problems and provide advice. Special thanks go to my Father and my Family for all their support.

Contents

Abstract	i
Acknowledgments	iii
List of figures	xi
List of tables	xiii
List of publications	xv
1 Introduction	1
1.1 Motivation and background	1
1.2 Research structure	3
1.3 Research focus and context	7
1.4 Thesis outline	10
2 Preliminary study: scientific software commercialization	13
2.1 Introduction	13
2.2 Methodology	13
2.2.1 Study participants	14
2.2.2 Data collection and analysis	17
2.3 Findings	18
2.3.1 The reasons for moving toward the development of a commercial pack- age	19
2.3.2 Software development considerations and issues during the transition to the development of a commercial package	20
2.3.3 Other considerations that arose during the transition to the development of the commercial package	24
2.4 Summary and discussion	26
2.4.1 The reasons for moving toward the development of a commercial package	26
2.4.2 Software development considerations and issues during the transition to the development of a commercial package	29
2.4.3 Other considerations that arose during the transition to the development of the commercial package	31
2.4.4 Conceptual findings from the preliminary study	33
2.5 Defining the focus for subsequent research	34
3 Literature review	35
3.1 Introduction	35
3.2 Contexts of scientific software development	35
3.2.1 Professional software developers developing for scientists	36
3.2.2 Scientists supported by professional software developers	37
3.2.3 Scientists as developers	38
3.2.3.1 Scientist-developers in High Performance Computing	38

3.3	Scientific end-user development	40
3.3.1	End-user development - definition	41
3.3.2	From end-user developers to developing for others	41
3.3.3	Tools and programming languages for end-user development	42
3.3.4	End-user development and scientific software	42
3.4	Recommendations for scientific software development practices	43
3.4.1	Recommendations for documentation	44
3.4.2	Recommendations for testing	44
3.5	Documentation in scientific software development	45
3.5.1	Reasons for not writing documentation	45
3.5.2	Issues with scientific software documentation	46
3.5.3	Scientific software documentation stakeholders	46
3.6	Documentation in professional software development	47
3.6.1	Documentation practices among professional software developers	47
3.6.2	Documentation in open-source projects	48
3.6.2.1	User-driven documentation in open-source development	48
3.6.2.2	Factors influencing documentation in open-source development	49
3.7	Documentation and software maintenance	50
3.7.1	What is software maintenance	50
3.7.2	Documentation and software maintenance	51
3.7.2.1	Source code and comments in code in software maintenance	52
3.7.2.2	People as sources of information in software maintenance	52
3.7.2.3	Documentation maintenance	53
3.8	Reasoning about documentation readers' knowledge	54
3.9	Communities of practice and scientific software development	55
3.9.1	Communities and network of practice	55
3.9.2	Knowledge exchange in communities of practice	56
3.9.3	Communities of practice and scientific software development	57
3.9.4	Communities of practice and scientific software adoption	58
3.10	Summary	59
4	Research design	61
4.1	Introduction	61
4.2	Research approach	61
4.2.1	Qualitative approach	62
4.2.2	Philosophical stance	63
4.3	Data collection	68
4.3.1	Study 1: semi-structured interviews	69
4.3.2	Study 2: contextual inquiry and work-based interviews	70
4.3.3	Study 3: case study	71
4.4	Ethical considerations	72
4.5	Data analysis	73
4.5.1	Inductive coding and thematic analysis	73
4.5.2	Quantitative analysis	78
4.6	Limitations	79
4.6.1	General limitations to a qualitative approach	80
4.6.2	Assessing validity	81
4.7	Summary	82

5	Study 1: scientific software documentation production	83
5.1	Introduction	83
5.2	Methodology	84
5.2.1	Participants	84
5.2.2	Data collection	90
5.2.3	Data analysis	90
5.3	Findings	93
5.3.1	Documentation in the scientific end-user development context	93
5.3.2	Documentation after the transition from the scientific end-user development to developing for a wider community	95
5.3.2.1	Documentation for software users	95
5.3.2.2	Documentation for oneself	97
5.3.2.3	Documentation shared with other developers in the team	97
5.3.2.4	Learning how to document	98
5.3.2.5	Perceived benefits from producing documentation	99
5.3.3	Documentation production and the user community	100
5.3.3.1	Perceptions of the user community	100
5.3.3.2	Feedback from the user community	101
5.3.3.3	Documentation and assumptions about users' knowledge	102
5.4	Discussion	103
5.4.1	Documentation practices in the scientific end-user development context	103
5.4.2	The community of practice and documentation production for others	104
5.4.3	Documentation production aids reasoning	105
5.4.4	The relationship between the user community and documentation production	106
5.4.5	Assumptions about users' knowledge	107
5.5	Summary	108
6	Study 2: scientific software documentation use	111
6.1	Introduction	111
6.2	Methodology	111
6.2.1	Study participants	111
6.2.2	Data collection	112
6.2.3	Data analysis	113
6.3	Findings	115
6.3.1	Scientific software documentation sources used by end-users	115
6.3.2	Scientific software documentation use by user-developers	119
6.3.3	Perceived advantages and disadvantages of documentation used by scientific software users	121
6.3.4	Documentation provided by the user community	126
6.4	Discussion	129
6.4.1	The rich variety of documentation sources	129
6.4.2	The main problems with documentation use	131
6.4.3	Trustworthiness of documentation resources	132
6.4.4	Documentation of the scientific model and details of its implementation	134
6.4.5	User community-generated documentation	135
6.5	Summary	136
7	Study 3: crowdsourcing documentation production	139
7.1	Introduction	139
7.2	Case study introduction	141
7.3	Methodology	143
7.3.1	Data gathering	143
7.3.2	Data analysis	145

7.4	Findings	147
7.4.1	Reasons for crowdsourcing documentation	148
7.4.2	Infrastructure for docstring crowdsourcing	150
7.4.3	Community’s expectations of the potential contributors	155
7.4.4	Standards and recommendations for crowdsourcing documentation	156
7.4.5	Engaging the community	157
7.4.6	Managing the documentation crowdsourcing process	161
7.4.7	Motivations for joining the documentation crowdsourcing project	164
7.5	Discussion	165
7.5.1	What may lead to scientific software documentation crowdsourcing?	165
7.5.2	Organisation and management of scientific software documentation crowdsourcing	167
7.5.3	The benefits of scientific software documentation crowdsourcing	168
7.5.4	The challenges of scientific software documentation crowdsourcing	169
7.5.5	Does crowdsourcing improve documentation?	170
7.6	Summary	171
8	Conceptual Findings and Discussion	175
8.1	Research summary	175
8.2	Emergent themes	176
8.2.1	Spectrum of users	177
8.2.1.1	Scientific end-user developers	177
8.2.1.2	Black-box users and white-box users	178
8.2.1.3	Other stakeholders: resource providers and collaborators	179
8.2.2	Spectrum of scientific software	179
8.2.3	Values	181
8.2.4	Reproducibility, reliability, and trustworthiness	182
8.3	Issues with documentation production in scientific software development	183
8.3.1	Insufficient resources	184
8.3.2	Lack of incentives for researchers to produce documentation	185
8.3.3	Keeping up with science	186
8.3.4	A community of practice - difficulties in changing practices	187
8.4	(Hidden) benefits of documentation production in scientific software development	188
8.5	Redefining documentation	190
8.5.1	Formal and “informal” documentation	190
8.5.2	Different roles of documentation	192
8.6	Considerations for crowdsourcing documentation	192
8.7	Significance of findings	193
9	Conclusion	195
9.1	Main findings and answers to the research questions	195
9.2	Further work	200
	References	205
	Appendix 1. Informed consent form	219
	Appendix 2. Preliminary study - interview questions	221
	Appendix 3. Study 1 - interview questions	223
	Appendix 4. Study 2 - interview questions	225
	Appendix 5. Study 3 - interview questions	227

Appendix 6. Coding schemes	229
Appendix 7. Data - excerpts from the interviews and mailing list archive	239

List of figures

1.1	Research structure.	6
2.1	Conceptual findings from the preliminary study.	33
3.1	Components of a social theory of learning	57
5.1	Conceptual findings from Study 1.	109
6.1	Conceptual findings from Study 2.	137
7.1	NumPy and SciPy documentation system workflow.	143
7.2	Timeline of the development of the documentation infrastructure.	152
7.3	Number of words edited in the SciPy and NumPy Documentation Project	159
7.4	Registered contributors to NumPy and SciPy Documentation Project	160
7.5	Screenshot: discussion under an edited docstring	162
7.6	NumPy installation packages downloads from SourceForge.	171
7.7	Conceptual findings from the SciPy study: stakeholders and their role in documentation crowdsourcing.	172
7.8	Conceptual findings from the SciPy study: considerations for documentation crowdsourcing.	173
8.1	Summary of scientific software features.	176
8.2	Conceptual findings: Scientific software spectrum.	181
8.3	Conceptual findings: Obstacles for writing documentation	184
8.4	Conceptual findings: Benefits from writing documentation.	190
8.5	Conceptual findings: Pattern of using formal/informal documentation	191
9.1	Research structure revisited.	197

List of tables

2.1	Preliminary study participants	16
2.2	Codes forming themes in Preliminary Study - software commercialization	18
4.1	Research approach vs. the type data	62
4.2	Examples of codes generated during the coding process.	75
4.3	Examples of refined codes	76
4.4	Example of merging codes in the data analysis for the preliminary study.	77
4.5	Example of merging codes in the data analysis for the study 1.	77
4.6	Example of merging codes in the data analysis for the study 2.	77
4.7	Example of merging codes in the data analysis for the study 3.	77
4.8	Example of grouping codes into clusters.	78
5.1	Study1 participants	89
5.2	Codes forming themes in Study 1- Documentation production	93
6.1	Study 2 participants	112
6.2	Codes forming themes in Study 2 - Scientific software documentation use	115
7.1	Study 3 participants	144
7.2	Codes forming themes in Study 3 - SciPy / NumPy study	147
9.1	Study 1. Summary of findings mapped to the research questions	198
9.2	Study 2. Summary of findings mapped to the research questions	199
9.3	Study 3. Summary of findings mapped to the research questions	200

List of publications

- A. Pawlik, J. Segal, H. Sharp, and M. Petre. Documentation practices in scientific software development. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2012 5th International Workshop on*, pages 113–119, 2012b. doi: 10.1109/CHASE.2012.6223004.
- A. Pawlik, J. Segal, H. Sharp, and M. Petre. Developing scientific software: The role of the internet. In *Science and the Internet, International Conference on*, Düsseldorf, Germany, 1- 3 August 2012c. URL <http://nfgwin.uni-duesseldorf.de/sites/default/files/Pawlik.pdf>.
- A. Pawlik, J. Segal, H. Sharp, and M. Petre. “Everything” about scientific software documentation that wasn’t in the manual. Keynote address at *Scientific Software Days, Austin, TX, USA*, 17th December 2012a. URL <http://scisoftdays.org/meetings/2012/>.
- A. Pawlik, J. Segal, H. Sharp, and M. Petre. Crowdsourcing scientific software documentation: a case study of the NumPy documentation project. *Computing In Science & Engineering*, . in print.

Chapter 1 Introduction

1.1 Motivation and background

Scientific software takes many forms, from modeling and simulation (for example, climate modeling and particle physics simulations), to manipulating and visualizing massive data sets (for example, genomics), to critical instrument control (for example, synchrotron, planetary exploration robots). The use and development of software in science has developed at an astounding rate, keeping pace with the development of the technology on which it relies and which has not only provided increasingly powerful hardware solutions but has also demanded increasingly sophisticated software solutions (Hannay et al., 2009).

At the same time, software development in science scaled up from individuals writing small programs (for example, solving a partial differential equation), to developed by groups of people producing massive code packages (for example, for complex simulations or analysing terabytes of data). Moreover, software is no longer stand-alone, but more often builds on suites of software developed by many people (for example, SciPy, libraries of Python code for scientific applications), and hence involves integration and re-use of software from many sources, often in new contexts.

Being a computational scientist means combining *advanced knowledge of software development* with *scientific expertise*. Computational science has two masters: science and software. And yet, for many, science remains the top priority, and software development remains secondary to the scientific goal, with little dedicated investment or attention to practice.

This is consistent with what is identified in the literature as “end-user development”, when people who are not professional developers with a task to accomplish in the world (such as sci-

entific analysis) develop software in order to support that goal. Their focus is on the domain, and software development is given a lower priority. End-user developers develop software primarily for their own use (or perhaps for use by a few colleagues). This is in contrast to “professional software development”, in which software development is the primary activity (and usually the work for which the developer is paid), and the software itself is the goal. The main difference between “end-user developers” and “professional software developers” is the goal: for the former, the goal is getting the domain work done, and for the latter the goal is developing effective software. It should be noted that this thesis considers a particular kind of “end-user developers”, “scientific end-user developers”. Their main goal is advancing their science, and software development serves that primary purpose.

A common perception is that, for a scientist, the technical knowledge required for the development of scientific software is easy to acquire. Programming does not require special attention, because it is merely a means to an end. Scientists developing software perceive their work as complete when they obtain the desired scientific results and, typically, publish the outcomes of their research (see for example, Segal 2009). Because software skills are not seen as having special value, and software development *per se* is not rewarded, the scientific community has not given much attention to developing effective software practices. And yet rigour in computational science also demands rigour in the software on which it relies, in just the way that other scientific instruments must be calibrated and understood. Flaws in software can lead to scientific disasters (Einarsson, 2005).

The need for further understanding of software development practices in scientific software has been discussed in the literature: in empirical studies (for example, Basili et al. 2008), in opinion pieces (for example, Wilson 2008) and in experience reports (for example, Baxter et al. 2006). In some contexts of scientific software development, software engineering tools and methodologies may be a poor fit. Teaching essential computing skills to scientists must be preceded by good understanding of which skills would actually be needed and useful. Providing guidelines well-suited to scientist-developers requires insight into their existing software development practices.

Voices from both research on scientific software development and the scientific community indicate clearly that there are a number of topics worth investigating in scientific software. For example, problems with reproducibility are widely discussed (for example, Stodden 2009 or Leveque et al. 2012); difficulties in reproducing published scientific results have led to a “credibility crisis” (Leveque et al., 2012, p. 13)). In computational science, **reproducibility** requires that the scientific software used to obtain the original results can be re-used by others. **Testing** is also a concern, which has been raised both in general terms about good research practice (for example, Hook and Kelly 2009) and in particular scientific domains, for example in astrophysics (Calder et al., 2004) and remote sensing (Hatton and Roberts, 1994). Many scientists do not know how to test their software and hence do not test it (Wilson, 2006). A related concern is the lack of an oracle to which to compare the outcomes of computational simulations (Hook and Kelly 2009), for example, in climate modeling (Easterbrook and Johns, 2009). Another concern is providing **relevant training** in computing skills and knowledge to scientists (for example, Wilson 2008). Members of scientific communities share their experiences, providing advice about the best practices related to scientific software development (for example, Heroux and Willenbring 2009, or Prlic and Procter 2012). The recommendations include advice on using source code management tools, tools supporting software development (such as issue tracking), tips on teamwork and much more.

The authors of these publications emphasize that addressing problems related to scientific software development should be one of the key priorities for the scientific community. Hence the motivation of the research presented in this dissertation is to understand current practice in depth (both what is going on and why) and to identify whether a software engineering perspective could offer any insight of practical value to scientific software development practice.

1.2 Research structure

As a first step in understanding current practices of scientific software developers, this research examined a key transition in scientific software, one which brings to light many aspects of the development process that may have been hidden previously: the transition from software written for use by an individual or a single group, to “commercialisation”, when the software

is used by others with their own goals. Such a transition has the potential to expose issues to do with assumptions, generality, reliability, and so on. This transition affects not only code, but also the scientist, whose role shifts from “scientific end-user developer” to “professional developer”. It also affects the primary goal, which shifts from furthering science to releasing software with commercial value.

The study of the transition from the scientific end-user development context to developing software for a wider community (discussed in detail in chapter 2) revealed a number of concerns, including adjusting the software to the requirements of the commercial market, dealing with intellectual property matters, and handling user feedback. In particular, the study highlighted the importance of documentation and of the practices related to producing and using documentation. This outcome helped define the focus for the remainder of the PhD research, and the main research question became:

How is software documentation produced and used in scientific software development?

The importance of documentation in scientific software emerges from the relationship between the science and computation. Documentation in science is not just about the software, but influences the application of the software to achieve scientific goals, given that reuse may involve a shift in the application. The information a scientist needs in order to reuse code does not all reside in the code itself, but also concerns:

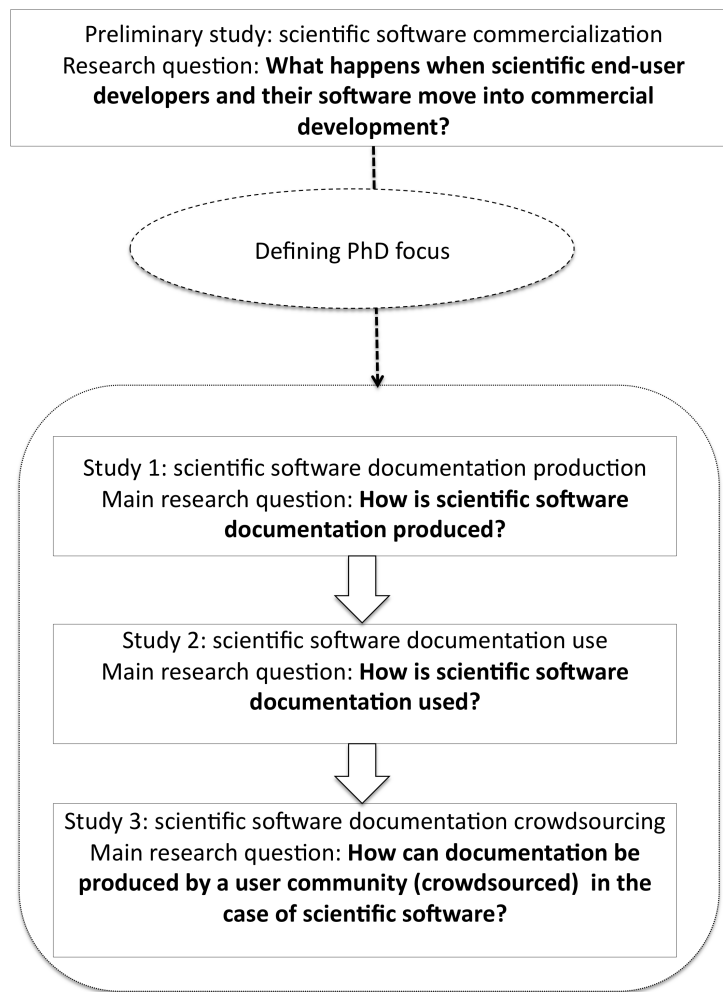
- the scientific method on which the code is based,
- implementation of the method in the software, which may involve numerical translation from a method in the world to something computable, and
- information about how to use the software.

The thesis shows that this information is exchanged in many ways. Therefore, the definition of documentation used in this thesis is quite broad, and includes any recorded, written information about software.

The three subsequent studies (studies 1, 2 and 3) each focused on a different facet of documentation. The research process was iterative and cumulative: each study informed the focus

of the study that followed it in a way that assembled a set of observations about the how documentation is created and used in scientific software. This is illustrated in the figure below.

Figure 1.1: Research structure.



The focus of study 1 was *how scientific software documentation is produced*. The preliminary study had confirmed that looking at the transition from “scientific end-user development” to developing scientific software for a wider user community was useful in revealing software development practices, and so the approach to look at the transition point was continued. Study 1 examined documentation production in both the scientific end-user context and the context of developing software for a wider community. A number of themes emerged, directing attention to the users and their relationship with the documentation. For example, scientific software documentation was often produced with an assumption that its users had a significant amount of science- and computing-related knowledge. These outcomes generated a number of questions about the relationship between the users and the documentation.

In order to answer these questions, study 2 focused on *scientific software documentation use*.

Study 1 showed that scientific software users can be represented as a continuum of users, from those interested only in software application (black-box users), to those who engaged actively with the source code (white-box users). Study 2 investigated how scientific software documentation was used by those different types of users. The most prominent outcome of this study was the role of the user community in documentation production. In all cases, the knowledge about the software which resided within the community was highly valued and useful. Very often this knowledge was captured in writing and thus became a form of software documentation.

Because study 2 highlighted the enormous potential of the user community to generate documentation, Study 3 investigated this topic further using a case study of documentation “crowdsourcing” done by a community around a suite of open source research software. The study looked in depth *how that community produced software documentation*, considering whether the potential to “crowdsource” documentation could be harnessed effectively. This final part of PhD research emphasised the theme that was present throughout the previous studies: the complementary relationship between software documentation captured in written form, and the ephemeral information exchange happening within the user and developer community.

The accumulated findings from all these three studies addressed the research questions. The richness of the collected data meant that there were also a number of emerging themes which not only enhanced the understanding of the role of documentation in scientific software but also opened a number of new directions for further research.

1.3 Research focus and context

This research, due to its exploratory nature, had a wide focus with respect to:

1. the scope of participants recruited for the empirical studies,
2. their scientific discipline and type of research, as well as
3. the kind of software they develop.

These are discussed in turn.

Scientists participating in the study

The scientists who agreed to take part in the empirical studies presented here had backgrounds in research disciplines ranging from crystallography and molecular physics to applied mathematics. All but four of the participants had no formal education in computer science or software engineering, nor extended professional experience of software development outside of academia.

The projects in which the participants were involved varied widely. The scientists were typically working on a number of different scientific software packages. In the interviews about documentation production and use (reported respectively in Chapters 5 and 6), the participants discussed their experiences across a number of projects, only occasionally referring directly to a particular experience from a particular project. The preliminary study (Chapter 2) and the study on documentation crowdsourcing (Chapter 7) each had a particular focus, and the participants were asked about their experiences relating to the software they commercialized (the preliminary study) or the NumPy package (the crowdsourcing study).

Participants' research scope

All but three of the participating scientists worked in an academic environment, either a university or a research facility that was largely involved in academic projects. The three exceptions worked in a commercial company or were self-employed. All participants had a postgraduate research degree (at least at the Master's level) in their scientific discipline. Their experience in conducting academic research varied from a couple of years (if they were still completing their research degree) to two or three decades of being independent researchers.

The research that the scientists conducted was in natural science disciplines. None worked in disciplines such as the arts and humanities (including social sciences). The participants typically had worked in a variety of research groups, from small teams including a few people to large multi-site collaborations.

Scientific software developed by the participants

In the preliminary study (reported in Chapter 2), the software developed by the participants was primarily in material sciences. One of the participants developed software which was used

for processing output data from a scattering device used primarily by physicists (of which he was one). The last empirical study (discussed in Chapter 7) focused on a large suite scientific library (NumPy).

The kind of software that the participants of the studies about documentation production and use (reported respectively in Chapters 5 and 6) referred to varied from short batch programs (a few hundred lines of code) to large software suites with a GUI and software libraries. The algorithmic complexity of the software also varied from highly advanced code implementing numerical methods or using complicated parallelisation techniques to less complex pipeline processing code. The programming languages used by the participants varied as much as the type of software they developed. In some projects they used general-purpose languages (such as Python, Java, C, or Fortran), and in other projects they relied on more purpose-specific languages (such as MATLAB). The application of the software depended largely on the discipline and, most of the time, it would be difficult to summarize in plain terms what these applications were doing. However, the lack of such description of the purpose of the scientific software developed by the applicants does not affect the data analysis and the research outcomes of this PhD. The main goal of the research presented here was to explore the role of documentation and the practices associated with it across domains, rather than tying them up to any particular context.

Software documentation

For the purpose of this PhD research, *software documentation* is understood and defined as: any written information about the software itself or about the software development process. The format in which such information is captured may be electronic (for example, a word processor document or a wiki) or paper-based (for example, handwritten notes in a notebook). This definition also encompasses non-textual forms of written documentation; that is, diagrams, graphics or screenshots used alone or in combination with text are also taken into consideration. It is also understood that those who use the documentation may be software users, other developers or the documentation authors themselves.

1.4 Thesis outline

The outline of this thesis reflects the research structure, as follows:

Chapter 2 - Preliminary study: Scientific software commercialization presents the findings from the preliminary study which looked at scientific software commercialization. This study helped to define the focus of the PhD research reported in this thesis.

Chapter 3 - Literature review presents the literature discussing scientific software development with particular focus on: the contexts of scientific software development, scientific end-user development, documentation in scientific software development, and communities of practice in scientific software development. Selected publications related to other research areas relevant to this dissertation are also presented. These include an overview of documentation production in professional software development and peer support in open-source communities. One of the sections looks at the mechanisms for reasoning about others' knowledge on a given topic (including the knowledge of documentation users).

Chapter 4 - Research Design presents the methodological approach to the research. Justification of the adopted qualitative approach is provided. Data collection methods and data analysis are discussed. The ethical issues and limitations are considered.

Chapter 5 - Study 1: Scientific software documentation production presents the findings from a study conducted with 22 scientist-developers, reporting on the production of scientific software documentation in two different contexts of scientific software development: the scientific end-user context and the context of scientific software development for a wider community of users.

Chapter 6 - Study 2: Scientific software documentation use presents the findings from a study conducted with 5 scientist-users, reporting on use of the scientific software documentation from the point of view of white-box users and black-box users.

Chapter 7 - Study 3: Crowd sourcing documentation production presents the findings from the case study of the SciPy/Numpy Documentation Project (documenting a suite of Python libraries for scientific software), exploring the theme of documentation crowdsourcing.

Chapter 8 - Conceptual Findings and Discussion brings together the findings from all the empirical studies and discusses them in relation to the relevant literature.

Chapter 9 - Conclusion and further work concludes on the research presented in this dissertation. Directions for further work are discussed.

Chapter 2 Preliminary study: scientific software commercialization

2.1 Introduction

This chapter discusses the preliminary study on scientific software commercialization. The study explored the transition from scientific end-user development to software commercialization. The focus was on the changes undergone by both the scientist-developers and their software packages. This topic of transition had not been covered in detail in the literature. Given the exploratory character of the study, the goal was to bring to light any issues that warranted further investigation. The study was guided by the research question: *What happens when scientific end-user developers and their software move into commercial development?*

The study provided valuable information which helped to define the direction and focus of the research presented in this dissertation, and this chapter presents the key findings. This chapter is structured as follows. First, the methodology used in the preliminary study is described. Second, the study findings are discussed in three main subsections: 2.3.1 “The reasons for moving from scientific end-user development to the development of a commercial package”; 2.3.2 “Software development considerations and issues during the transition; 2.3.3 “Other considerations that arose during the transition”. The third section discusses these findings, referring back to the literature (chapter 3) where appropriate. The final section describes how the focus for the subsequent research was defined.

2.2 Methodology

Understanding what happens during the context change from end-user development to commercial development required collecting stories about commercializing scientific software from those who actually did it and were able to reflect on their experience. Studying practice in real-

life settings is considered to be one of the best ways to understand a whole variety of issues in software development (Robinson et al., 2007).

2.2.1 Study participants

Recruiting the participants posed some challenges. First, participants had to fit the main criterion of being scientists who themselves transformed their research software into a commercial package. Second, the focus on proprietary software meant that some potential participants were afraid of revealing details which could have hindered their business. Seven participants were recruited via convenience sampling; personal connections proved to be the most effective way to overcome the challenges. All participants had a research degree in a scientific domain: two in Physics and five in Material Sciences and/or Metallurgy. The core educational background was roughly similar for all participants: they all had knowledge of higher Mathematics, Physics and Chemistry. The six software packages which they created were very specialized. Five packages were from Material Science, and one was from Physics.

Five participants worked in five different companies, which were either founded or co-founded by them. The other two interviewees worked together in the same company, which one of them established (and the other one joined some time after its foundation). In other words, the seven participants came from six different companies. All participants underwent a transition from being a scientific end-user developer to developing commercial software for a larger group of users.

In the case of six interviewees, the transition from end-user development toward developing for a wide group of users occurred during the process of commercializing the scientific software which they initially built for their own use or that of a few lab colleagues. The seventh interviewee did not commercialize his own software package but joined one of these six companies. Moving into a commercial environment meant not only developing for a wider community of users, but also adding the goal of making a profit. It should be noted that out of the six participants who founded or co-founded their companies one was not engaged in his company any more at the time of the interview. However, the other five were combining the roles of businessman and commercial software developer with the role of scientist, occasionally using their

own software in order to conduct research.

None of the participants had a formal degree in software engineering or computer science none had any more formal training than one- or two-semester courses in programming during their university studies. All interviewees said that they had acquired most of their knowledge about software development from a variety of informal sources: books, online resources, and advice given by their colleagues. The table below presents information on the interviewees together with an overview of the commercialized software packages.

Table 2.1: Preliminary study participants and overview of the software commercialization cases

ID	Domain	Commercialization timeline	Commercialization model	Location
A	Material sciences and metallurgy	The software was commercialized in the late 1990s.	Establishing a partnership with a lab colleague. In this case there were no major issues with establishing the ownership of the intellectual property rights.	Eastern Europe
B	Material sciences and metallurgy	The software was commercialized gradually over the 1990s.	Initially setting up a foundation which held copyrights to the software and then transforming it into a commercial company.	Western Europe
C	Material sciences and metallurgy	The software was commercialized in the early 1990s.	Initially a spin-off company funded by the Ministry of Economic Affairs. Then the company became fully independent.	Western Europe
D	Material sciences and metallurgy	The software was released as a full commercial product in the early 2000s, however as the interviewee put it: “[since 1975] The whole thing has been hand in hand - the university research and the development of the commercial software”.	Setting up a company and commercializing software was highly encouraged by the university. The profits from the commercialization came back partly to the university and partly to the scientist.	North America
F	Material sciences and metallurgy	The software commercialization started in approximately mid 1990s. The company became an independent organisation in the late 1990s.	Initially setting up a company solving problems for a consortium of commercial companies. Then the firm started selling the product more widely and became independent of the consortium funding.	Western Europe
F	Physics	This interviewee did not commercialize his own software but decided to leave academia (after completing two post-doc contracts) and change his professional to being a software developer (specializing in scientific software).	Interviewee F joined the company established by the interviewee E.	Western Europe
G	Physics	The software was commercialized in the early 2000s. Before that it was used for about a couple of years internally at the scientist’s university.	Establishing an essentially one-person company. The only issue mentioned by the interviewee was sorting out the intellectual property ownership - the software is available free to anyone from his university.	North America

2.2.2 Data collection and analysis

The exploratory nature of the study meant that the data collection method should not restrict the topics that might emerge from the research. Semi-structured interviews were used, in order to gather rich and descriptive data while providing a common framework of questions. (Robson, 2002). All interviews were conducted via phone, audio-recorded, and transcribed. The transcriptions were then analyzed using bottom-up thematic analysis (Boyatzis, 1998). The interviews were built around three main questions:

1. What was the story of the commercialization of the software?
2. What were the challenges during that commercialization process?
3. What appeared to be easy or worked well during that commercialization process?

In the course of the interview, additional questions were asked in order to clarify points or to encourage the interviewee to elaborate on a topic he brought up. The analysis process revealed a plethora of topics and issues related to the process of the transition from end-user developer to developing for others.

During the analysis, small parts of the transcribed interviews were coded inductively (that is, allowing the codes to emerge from the data). The codes were then used to form the themes:

Theme	Codes
Reasons for moving toward the development of a commercial package	Commercial-Funding-for-Research; Difference-Commercial-Science-Academic; Software-Academic-Phase-Sharing; Software-Commercialization-Research; Software-Academic-Phase-Obstacles; Software-Commercialization-Start-Reasons; Software-Commercialization-Creating-Company; Software-Commercialization-Funding; Software-Start-Funding;

Software development considerations and issues during the transition to the development of a commercial package	Academi-Phase-Code-Sharing; Software-Development-Now-Problems; Software-Developments-Past-Difficulties; Software-Commercialization-Phase-Changes; Software-Development-Anticipation-Technology; Software-Development-Anticipations; Software-Development-Milestones-Technology; Software-Development-Stages; Software-Performance-Technology; Software-Development-Parallel-Architectures; Software-Development-Plans-Future
Other considerations that arose during the transition to the development of a commercial package	Software-Development-Combining-Academic-Commercial; Software-Commercialization-Problems-Marketing; Intellectual-Property-University-Software-Profit; Software-Commercialization-IP-Licensing; Software-Commercialization-Copyrights; Software-Commercialization-Finding-Customers; Software-Commercial-Phase-Company-Development; Career-Change-Toward-Software-Development; Software-Commercialization-Change-Team; Software-Development-Collaboration; Software-Commercialization-Finding-Partners; Software-Development-People-Background;

Table 2.2: Codes forming themes in Preliminary Study - software commercialization

2.3 Findings

The findings presented in this section are grouped under three main topics: i) The reasons for moving from scientific end-user development to the development of a commercial package are discussed. ii) The software development considerations and issues encountered during the transition from the scientific end-user development context to the development of the commercialized package. This section also presents findings about the software development practices reported. iii) The considerations and actions not related directly to software engineering and development which the participants met when transforming the software from its original version into a commercial package.

2.3.1 The reasons for moving toward the development of a commercial package

The study explored the reasons for scientific end-user developers to move toward developing commercial software. Financial benefits are the most obvious reason, but the participants also reported a number of other reasons, which shall be discussed in turn.

One reason was the attitude of the research organizations at which the scientists worked. Two participants (both from North America) said that their universities actively encouraged the academic staff to undertake entrepreneurial activities. It was an ingrained assumption that anything they built in order to advance their research should also be considered as a technology useful for industrial application.

The initial motivation for developing the software was to support the participants' own research. Thoughts about making the software commercial came later.

I had a number of other experiences in the past of writing software for analysis of some sort. In all those cases I didn't see any reason to go any further but this time [it] seemed to me that it was useful enough the thing that I had written that I thought I should convert that into a commercial product [Interviewee G]

Those participants who came from European universities did not report receiving encouragement and support from their universities for making the products of their research activities commercial. One participant said that the decision about commercializing the software was made externally by the department at which he worked in collaboration with the funding body. However, this decision was made when the research software was already quite well developed and showed some potential to be used outside of the laboratory in which it originated.

Another reason to enter the commercial market was the interest that the software was receiving from users who had gotten hold of an early copy of the code. Four interviewees mentioned the common practice of free code or program distribution and exchange during the phase when software was developed at the university with no anticipation of further commercialization. However, even if the software was used by more than just its developer, the participants' attitude toward their software was still very much that of a scientific end-user developer. That is, the software was treated as a means to an end rather than the end itself. It should be noted that all except one of the software packages in the study discussed here originated in the late 1970s and 1980s, when the means of informal source code dissemination (typically, via floppy disk) were much more limited than nowadays (e.g., downloading from an online source).

Nevertheless, the scientists-developers did not have full control over how the code was shared and how information about it spread within the scientific environment. The inquiries about the software sometimes came from unexpected sources (compare: Quote 2-1). This external interest in the software made

the participants, who at that point were still scientific end-user developers, realize that their program had the potential to become a commercial package.

Summarising, there were various reasons for commercializing university-developed scientific software. External interest from research organisations who had learnt about the software informally was one reason that prompted the idea of commercialization. The general pro-commercializing attitude at the participants' universities was another. A direct decision about starting a spin-off taken by the organisation which originally funded the research was a third way of initializing the commercialization.

2.3.2 Software development considerations and issues during the transition to the development of a commercial package

In all cases, the software was initially developed without the expectation that it would be used by multiple users. None of the scientists reported working extensively on the design of the software, and none mentioned detailed or rigorous methods of gathering and analysing requirements. Software development appeared to be an organic process, and therefore there was no planning ahead supported by explicit work on the design. One of the interviewees, when asked directly to clarify what the main difference was between designing software under DOS and then Windows, replied: "What do you mean by design?".

Another scientist said that lack of upfront design turned out to be problematic in due course. The main problem reported by the interviewees was that expanding the software and adding new functionalities became extremely complicated due to difficulties in recalling implementation rationale and details (compare: Quote 2-2).

The initial lack of design, planning, and structuring also affected the way the participants fixed bugs. Because the software design was typically not in place from the beginning, any changes in the code had to be somehow just "fitted" into the existing structure. The process of fixing bugs and resolving problems was often limited to just applying a "patch", as the interviewees were unable to reconstruct what had actually been done in a particular piece of source code which was causing problems. They even had difficulties identifying where in the source code the issue really was.

In addition to issues due to lack of design, the interviews revealed that the documentation for the software was typically scarce.

*we just produced the software and didn't put annotations or notes somewhere else.
The only thing we wrote down was help pages. [Interviewee E]*

In one case, the interviewee explicitly said that some elements of the code were developed solely by a very experienced scientist who not only had a very advanced knowledge of the particular scientific method but also remained the only person responsible for this part of the code. He did not provide

extensive documentation for the part of the software which he developed. In the same project, there were a few professional software developers employed on a short-time basis to complete selected tasks. The interviewee was then able to compare the documentation which they produced to documentation provided by the scientist.

A different interviewee had a similar experience of dealing with pieces of source code which were understood only by the scientists who wrote them.

there are people who are retiring who wrote the code 20 years ago. Nobody else can understand it except them. Now we're trying to re-write some of that and talk to these people and get the things structured properly, commented and documented. [Interviewee D]

None of the interviewees reported using any methodology or system for creating documentation. The practice of archiving the documentation that did exist, such as user manuals for successive versions of the software, was reported by one of the participants as “pointless”. During an informal discussion with another participant, it became clear that he and his colleague (who run the company) did not store the software documentation in any organized way. The participants unanimously reported that scarce technical documentation eventually caused serious problems with software maintenance. One of the participants said that not producing documentation only *seemed* to be time-saving. A lot of time was wasted due to lack of documentation (compare: Quote 2-3).

A different participant commented that over time the code reached the “spaghetti” stage. It was confusing and complicated, due to lack of planning and not keeping the structure “general” enough so that it was relatively easy to re-develop in the future (compare: Quote 2-4).

The participant said that the whole team involved with the software thinks that they should try to address the problems caused by lack of documentation (including issues with understanding “spaghetti code”) by simply rewriting the whole software package or selected parts. However, he commented that it was a massive task, and it would be difficult even to decide where to start. Another participant had a similar idea to rewrite the whole software to make the source code more “comprehensible”. He also perceived this as a task which would be incredibly difficult to complete.

It should be noted that one interviewee had quite different experiences with regards to design and documentation. From the very beginning he decided to write his software in Java, and he emphasized that during the development he tried to stick to the recommended Java standards, and with every new Java release he updated his software accordingly. He also used JavaDoc in order to generate documentation automatically from the source code and developed a rigour in maintaining documentation.

All participants reported regular software releases which contained new functionalities and fixed known bugs; however, no one mentioned using version control. One interviewee said that he simply exchanged source code files via email with his co-developer, a practice which sometimes resulted in problems with being able to merge newly developed features smoothly into the new version of the software. Another scientist explained in detail how he kept track of the changes in the software.

I could manage basically all by myself and in principle new features which were added either I added right away or they were forgotten and maybe they had to come back next year and ask for the same addition again. There was very little documentation except that each year we had the frozen version which we could just look at and check things how it looked a year or two before. We started on version E...we have the version E, F, G, H, I J K L M N Q P R...we're currently at version S. So one can go back and run version M and find how that worked. [Interviewee B]

The scientists reported that transforming their software into a commercial package required developing new modules (like a GUI) in various programming languages. In five cases, the core part of the software was written in FORTRAN combined with C, Delphi, C++, and so on. In one case the whole program was written in Java using an object-oriented approach. In general, the participants were familiar with the concept of object oriented programming. One interviewee said that, when he was developing the software at the very beginning, he read one of Donald Knuth's publications¹ and, inspired by some of the "pre-concepts of objects", designed some of the data structures in an "object-like" way.

The participant who used Java to build his software was very positive about the decision he made and perceived deploying Java to be easy and beneficial. Even though he did not have prior experience in object-oriented (OO) development, he decided to use this paradigm for his software. Knowledge of Java and OO was acquired through self-teaching and, according to the interviewee, it was much easier than learning and programming in FORTRAN (compare: Quote 2-5).

Adapting the software for the needs of a commercial market not only meant making it user-friendly, it sometimes also involved porting the software from a mainframe to a PC platform. Additionally three participants emphasized the importance of the advent of WIMP (Windows, Icon, Menu, Pointing device) interface-based systems (like Windows). Despite the fact that they had to invest a significant amount of effort into redeveloping their software so that it could be used on PCs and with WIMP systems, they all said that their commercial success would have been practically impossible without the ubiquity of these technologies and the market opportunities they created.

I think if PCs had not emerged then we would really not exist because we need the big market out there. We need all those university institutes and also the research

¹The participant refers to Donald Knuth's publication now available as a book - "The Art of Computer Programming", Addison-Wesley Professional; 2011

labs in the industry. Otherwise we don't have a wide enough customer-base so that you can run a company based on that. But of course, without keeping an eye on the scientific development in our field you will also sooner or later falter. Both are necessary. I would think scientific development alone...well, OK if everybody else in the field could not offer the PC software, then it would be the same for all our competitors and there are not very many, about half a dozen. They would all have this difficulty of not having a sufficiently large market from which you could actually get sufficient income to run a company. So I think having PCs is really essential. [Interviewee C]

The software developed by the participants is highly specialized, and their customers constituted a relatively small community, counted in hundreds rather than hundreds of thousands. However, one of the interviewees to his surprise discovered that somebody prepared and released a “crack” for his software.

I had to completely rewrite the key generation, the security. I had to get much more serious into learning the cryptography, learning how to produce keys and so forth. I hadn't seen that [until] then and I still don't know whether those sites were genuinely interested in releasing the cracks for the programme or were they simply those sites to produce cracks for everything, and all they were really trying to do was put spywares and viruses on your computer. But it was upsetting. I realized I was a little naive about the environment and the internet. [Interviewee G]

The transition into commercial development introduced dealing with increasingly frequent clients' feedback, which did not exist at all before the transition. Obviously users' needs and requests had to be treated with attention, because in order to boost and maintain the sales the participants had to ensure that the software did what their clients wanted. Users' feedback was encouraged. This feedback could range from reporting bugs to suggesting or even partially implementing new functionalities for the software. Handling the customers' feedback was considered important and was often dealt with by several people (in four companies, in addition to the core development team, there were also sales and support staff). New functionality requests were usually prioritized according to their frequency and potential usefulness for other customers.

In summary, several software development considerations were reported by the participants. In the initial stages of the software development, any planning and designing of the software was limited or non-existent. In the later stage, when the software was commercialized and new features were added, this lack of planning and design lead to problems as the scientists struggled to recall implementation rationale and details. A similar problem occurred with bug fixing. Documentation, in particular technical documentation, was scarce, which added to the problems resulting from the lack of design and planning in the initial stage of software development. The changes to the software required by the commercialization process highlighted that some significant parts of the source code were known and understood

only by the scientist who developed it but who was not around any more. Complete source code rewriting, in order to make it more maintainable, was considered by at least two participants. None of the participants reported using version control. Two of them described their own approach to versioning the software. Learning and using different programming paradigms (such as object oriented programming) and acquiring skills in software development was another thing that was taken into account by the participants. They were all largely self-taught. In the cases when commercialization happened at the same time as the rapid development of PCs and GUIs, adapting the software for these new technologies was a consideration. Securing the software from unauthorized use was reported as a concern during commercialization. Finally, handling users' feedback had to be taken into account.

2.3.3 Other considerations that arose during the transition to the development of the commercial package

The participants typically started developing software on their own or within a small group of lab colleagues. Redeveloping software so that it could be introduced into the commercial market sometimes required new developers to be taken on board.

There were already two of us, when the software was redeveloped from the DOS to the Windows version. I asked my colleague to join me once I sold the first DOS versions. [I did that] because it turned out that designing, redeveloping and doing a lot of things from scratch which occurred once it became clear that Windows allows you to create a GUI (...); it turned out that in order to be able to do it so that this product could become sellable within reasonable time (...) so that the customers were not left disappointed (...) it was too much for one person to do.
[Interviewee A]

Another participant reported hiring new developers on a short-term basis, to complete particular tasks.

We started off hiring somebody who was good at the client end, in other words he could produce the user interfaces but he wasn't necessarily [a] material scientist which is what my colleague and I were. We then had one other person who was another material scientist and whom we could give jobs to do. I think at one point we had a third person, a mathematician, to help us streamline the algorithms.
[Interviewee E]

In those cases in which developers without scientific background were hired, they were assigned "limited" and well-defined tasks (like building a more sophisticated GUI or input-output handling module). The new developers employed on a longer-term basis or invited to be business partners were scientists from the same or a very close scientific domain who had had some experience with software development.

Introducing some changes to the software which were related to adjusting the scientific model underpinning the software required a relevant level of domain knowledge. The programmer would need either to

have experience in handling such models or to at least understand them (compare: Quote 2-6).

Where mentioned, division of labour was described as easy to achieve and was not reported to be a source of conflicts. In the cases where there was a small core team of two to three initial end-user developers over the years, they established the division of task and work organisation which was most convenient to them. This division of labour and responsibilities was achieved via implicit negotiation. Essentially, everyone did what they liked and what they were good at, and, as one interviewee put it: “luckily their preferences did not overlap”.

However, this straightforward and stable division of labour meant that code was rarely shared. Only the developer who worked with a particular piece of code was able to understand how he implemented selected parts. This would eventually lead to problems when handing the software over to those who would take over the companies when the original creators of software retired.

All but one of the participants also had to sort out intellectual property (IP) matters. The exception said that he did not have to discuss the IP with the employer, because the software he developed was not directly related to the work he was doing at his research organization. The others had to go through some paper work and negotiations; however nobody reported having any particular difficulties.

Six interviewees still worked at their research organisations after commercializing their software. One participant in particular discussed the trade-off that scientific end-user developers may face when they decide to commercialize their software. He said that the development of new modules or functionalities was almost entirely driven by the customers’ requests. According to the interviewee, at the creation of the company, he and his colleagues produced academic papers focused on the theoretical aspects of their work. Later on they moved on to publishing papers which had a more practical angle.

Despite having had between ten and twenty years of experience in developing commercial scientific software, most of the participants still perceived themselves primarily as researchers rather than entrepreneurs or professional software developers (compare: Quote 2-7).

Only one participant had a different perception of himself.

I’m considering myself not really a scientist. I’m an engineer. I solve problems. I don’t really try to understand what is the problem. [Interviewee F]

The combination of two roles, that of being a researcher and that of being a software developer and/or an entrepreneur, was not reported as posing major challenges. Background information about the interviewees and the data obtained from the interviews indicated that they balanced academic and commercial activities.

In summary, in addition the software development considerations which arose during the commercialisation transition, there were other matters which had to be taken into account. In some cases, additional developers had to be hired, requiring consideration of what the desired knowledge and background of these developers would be. The professional software developers employed who did not have background in the scientific domain were usually assigned a self-contained and well-defined piece of work (such as developing the user interface). For changes in the software which were related to the science underpinning it, it was important to hire developers who had a relevant background and expertise in science. In cases in which the development team was expanded, the division of labour had to be considered. All but one participant reported dealing with intellectual property matters. Finally, combining being a researcher and being an entrepreneur does not seem to have been thought through consciously. The software commercialization process resulted in balancing between the two worlds: academia and commercial software development.

2.4 Summary and discussion

This section summarizes the reported findings and discusses them in the light of the literature. It should be remembered the study's goal was to gain broad, rather than in-depth, understanding of what happens when a piece of software developed originally in the scientific end-user development context is commercialized. The themes discussed in this section revealed a number of questions and helped to define the focus for further research reported in this thesis. First, the reasons for moving from the scientific end-user development to the development of a commercial package are discussed. Second, considerations and issues related directly to software development are discussed. The last subsection covers other considerations, not related directly to software development, that arose during the transition from the scientific end-user development to the development of a commercial package.

2.4.1 The reasons for moving toward the development of a commercial package

Due to the exploratory character of this preliminary study, The publications considered in the following paragraphs are referenced in order to indicate how the findings from the preliminary study relate to existing research. The discussion does not span the wider literature about commercialization of academic projects and technology transfer, which is outside the scope of the preliminary study.

The participants reported several reasons for commercializing software originally developed in the scientific end-user context. Similarly, the literature points out that there may be various motivations for commercialisation of research outcomes. D'Este and Perkmann (2011) identified four independent motivations: financial benefits, learning, access to in-kind resources, and access to funding. Commercialization may be seen simply as a mechanism for gaining financial benefits from engaging with industry.

Alternatively, it may facilitate learning, receiving feedback and information on new problems from industry, checking the applicability of the research as well as joining a wider network. It may afford access to in-kind resources, such as materials, equipment and additional expertise. It may provide access to additional funding from industrial collaboration, which can be spent on academic research. The findings from the preliminary study presented in this chapter indicate that the motivation for commercialization may come from an “external” source. That is, the idea of commercialization may emerge after there is some concrete interest in the software from industry, or when the scientists’ research institution decides that commercialization is desired.

The participants of this preliminary study did not report access to in-kind resources or access to funding as motivations. However, one of the interviewees mentioned that there was an idea of altering the financial structure of the company, so that part of the commercial income would fund research work. But, as the interviewee indicated himself, it was only an idea, and no actions toward establishing that kind of scheme were undertaken.

As D’Este and Perkmann (2011) note there may be various ways in which academics may engage with the industry. These include the creation of spin-offs, patenting, consulting, contract research and joint research. They discovered that there was a relationship between motivations for engagement with industry and the ways of engagement. Commercialization had the strongest positive correlation with creating spin-offs and patenting. In all cases in the study reported in this chapter, interaction with the industry meant setting up a company. Not all of them were spin-offs created within the environment provided by the original research institutions. Nevertheless, other kinds of interaction with industry were not reported. Patenting computer software has been a debatable issue, and in most countries it is not possible to patent a computer program¹. Therefore it is not surprising that the study participants did not discuss it.

Lam (2011) suggests that the motivation for commercializing research outcomes may differ, depending on scientists’ perspectives on university-industry links. Lam identified four types of scientists’ approaches which represent a kind of continuum: a pure traditional who believes that academia and industry should remain distinct; a pragmatic traditional who accepts that occasional collaboration is possible for pragmatic reasons; a hybrid who is convinced that collaboration is useful but that academia should remain independent; and an entrepreneurial for whom collaboration between academia and industry is essential for knowledge exploitation and application. Two of the participating scientists based in the Northern American universities be described as hybrid or entrepreneurial, or at least the approaches of

¹Software cannot be patented under the European Patent Convention (for details see <http://www.epo.org/law-practice/legal-texts/html/epc/2010/e/ar52.html> - Retrieved 25-08-2012). Software is not explicitly listed as patentable under the U.S. patent law (for details see http://www.uspto.gov/web/offices/pac/mpep/documents/appxl_35_U_S_C_101.htm#usc35s101 - Retrieved 25-08-2012)

their research institutions could be described in these terms. What remains unknown is the extent of the influence of the institutional approach on the individuals' approaches and attitudes toward commercializing research outputs.

The original motivations and attitudes toward commercialization may have further impact on the development of such undertakings (Lam, 2011). Some commercialization models (such as spin-offs) require scientists to make decisions related to managing people, budget, marketing, production, and so on. As Gurdon and Samsom (2010, p. 210) conclude: "the technology is a necessary but not a sufficient condition for success. It is quite clear that both scientific excellence and management expertise are critical in explaining the performance (...)". Indeed, as it is discussed further, the findings of this study show that there were a number of non-software development considerations which emerged during the commercialization process. The literature indicates that those scientists who, along with developing their business, carry on with their academic careers experience a tension between "the culture of business and the culture of science" (Gurdon and Samsom, 2010). Such tensions have not been explicitly reported in this preliminary study. One participant said openly that, after spending several years in academia, he decided that this culture did not fit him, and he then changed his career, leaving the academia and becoming a software developer (specializing in scientific software). He actually did not leave academia during the commercialization of his software. But he explained that the characteristics of working in academia (the recognition system, bidding for grants, and so on) did not suit him at all. According to him there was a big contrast between working in academia and in a commercial environment.

Lam (2011), in her study of UK-based scientists, discovered that the majority of researchers hoped to enhance their scientific careers and build their reputations through commercialization, which provided much-needed funding to support their research projects (which is similar to the observations made by D'Este and Perkmann 2011). According to Lam's findings, increasing personal income was a much-less-frequently-occurring motivation. The participants of this preliminary study did not actually report if the financial benefits from commercializing their ideas helped to support their scientific activities. Once their software became a commercial package, its further development was largely driven by customers' requests rather than advancements in science. But Lam also noted another, more intrinsic motivation, quite common for those scientists in the preliminary study who decided to engage with industry. It was "the sense of achievement that they [the scientists] experienced in starting up a business" (Lam, 2011, p. 1365). At least one of the participants of the study reported in this chapter shared the same feelings as he clearly expressed satisfaction with the fact that his business succeeded.

The findings of this study indicate that sharing the code in the scientific end-user context resulted in the growing popularity of the software, which, eventually, spurred the idea of commercialization. In some cases, the distribution of the copies and of the dissemination of the information about the program was out of the control of the original developers, and they did not know who received a copy of their software. However, it turned out to be for their benefit. Some of the copies of the software were apparently noticed by some commercial companies. The positive feedback and direct business inquiries prompted them to think about commercializing the software. In the case of this preliminary study, the software was made available in the scientific end-user context to other users, but this process was quite informal (a copy of the software was given upon someone's request). The change into the development of a commercial package involved a number of different considerations which are discussed further in the next section.

2.4.2 Software development considerations and issues during the transition to the development of a commercial package

The software considered in this preliminary study was originally developed to support the participants' own scientific work. They typically developed the software on their own or in a small group of other scientists, with little design or planning. This finding is consistent with the results reported by Hannay et al. (2009). Out of 1972 respondents 58% developed software on their own, and 35% worked with one more person (but in teams smaller than 6 people in total).

It is also possible that the lack of design or requirements documentation was related to the difficulty of capturing and defining requirements up-front, rather than allowing them to emerge during the research process (of which software development is a part). In many, if not most disciplines, software development is a part of the exploration and discovery. The requirements are emerging (Segal, 2008a). For some scientists, the concept of a requirements document may be completely useless (Segal, 2005b).

As participants reported, the lack of planning and structuring of the software caused problems later. Software maintenance and in particular extending the software's functionality was problematic and time-consuming, because the software was not designed to accommodate changes at a later stage. The problem was exacerbated by the application of patches to fix bugs in the source code. This approach is similar to the practice observed among scientist-developers of using software solutions which are "good enough" to address a given problem at a given moment (Segal, 2007) and not necessarily considering any future use of the software.

Difficulties also arose because different parts of the source code were written by different scientist-developers, and code sharing was almost non-existent. Sharing knowledge within the team was rare.

The fact that everybody did what they essentially liked in software development seemed an advantage at the beginning, but resulted later in serious issues. Maintenance of the software became a significant problem. With one exception, the interviewees did not mention using any coding style guidelines, and it is possible that each one of them had their own coding style which eventually contributed to difficulties with understanding the code written by other developers. And as de Souza et al. (2005) found, source code and code comments are the two most important documentation artifacts for software maintainers. In general, this preliminary study of the scientists provided the desired expertise and work effort, however the exchange of experiences and mutual learning was on a level which did not suffice for creating maintainable software.

One of the participants who commercialized his software, developed in Java in the early 2000s, did not report major problems with documentation. He reported that he used a tool supporting automatic documentation generation. He was also the sole developer of the software and hence he had to capture enough knowledge for himself to understand the implementation details in the future. He reported that he developed a certain rigor in software development which he maintained throughout. The initial effort and investment was relatively high, but it was clear to him that it paid back. It should be noted that this scientist had taught himself object oriented programming and Java. He did not report having major problems with acquiring this knowledge. He even perceived Java as being easier to learn than FORTRAN or C which he used before. The fact that the participant did not have problems learning a generic programming language echoes what (Segal, 2005a) noted (as discussed in detail in chapter 3). Generic programming languages (such as Java or C++) are also used in scientific software development done by mixed teams consisting of scientist-developers and professional software developers (for example, Morris 2008). However, it should be noted that domain specific languages may be preferred and used by scientists (for example, Jones and Scaffidi 2011). It is possible that the preference to the use of domain specific or generic programming languages may be related to the domain.

Participants were largely self-taught with respect to software development; they reported using a variety of programming languages in their projects and seeking different solutions for designs of their software. The scientists developed their own approaches to software development which addressed their needs in their particular contexts. For example, one of the participants described his own way of archiving different versions of the software without using a version control system. This resonates with the findings reported by, for example, Easterbrook and Johns (2009) that scientists working on the same application for a period of time find their own solutions and strategies in software development which suit their unique situation and needs.

Commercialization of the software required the scientists to perform a number of changes which would not have been required if the software had remained in its original context. Transformation of the software so that it was compatible with other operating systems and was user-friendly were the key moves in commercialization. When the software was used in a limited context, by the scientists themselves or their close colleagues, the inconveniences of the lack of a GUI or the necessity of running the software on different machines were not issues. The case of the participant who had to make his software more robust against hacking attacks illustrates the different requirements which the software has to meet when shifting from the scientific end-user context to a commercial market. Kelly et al. (2007) noted that the tests revealed the necessity for some in-depth re-engineering to be applied on the code to make it a commercial package. There is a clear indication that commercialisation may pose challenges which would never arise if commercialisation did not happen.

Software commercialization also means that changes in the software package are subsequently driven by the needs of the customers, rather than the research needs of the developer. The aim of the scientist-developers to progress their own science and explore new possibilities may become secondary to requests coming from the users. The decisions about which new features should be implemented first are not motivated by, for example, new discoveries reported in the relevant discipline but simply by business goals such as satisfying a large number of customers in one move. Handling the users' feedback and planning new development becomes a key element of software development, one which was not present in the scientific end-user context.

2.4.3 Other considerations that arose during the transition to the development of the commercial package

Commercializing software brought up considerations in hiring additional developers. Further work on the software, in order to commercialize it, required understanding of the science. This observation is consistent with the one made by Carver et al. (2007) who note that "Domain complexity is evident in the fact that much of this software is written to simulate highly complex physical or engineering behavior. In fact, many of the applications require a PhD in physics or a branch of engineering to understand the problem. The teams have found it easier, and more practical, for the domain scientists and engineers to learn how to write software than for software engineers to learn all of the relevant science and engineering concepts" (Carver et al., 2007, p. 558).

On the other hand, for specific tasks such as improving the performance or working on the GUI, it made more sense to hire for the short-term someone who did not necessarily know the given scientific domain but who had skills and experience in dealing with similar tasks. Indeed (Carver et al., 2007, p. 558)

found that “to achieve performance and flexibility in such complex applications, the teams also are in need of software engineering expertise”. Aranda et al. (2008) explicitly make a distinction between technical human resources and scientific human resources within the teams they investigated (as discussed further in chapter 3).

Since none of the projects described in this study employed professional software developers employed long-term, it is difficult to assess how the collaboration between them and the scientist-developers might have developed. Collaboration between these two groups may pose major challenges (Segal, 2009). However, Segal’s study focused on software developed in a research-academic context rather than in a purely commercial setting. Scientists focused on advancing their research, exploring the domain with evolving code and obtaining publishable results, while professional software developers wanted to develop software meeting the requirements which had been defined upfront. The dynamics of this interaction may change when scientists-developers (like the participants in the study) move on to commercial software development. The findings reported in the previous section indicate that transition to the commercial context introduced a number of changes to work practice and priorities.

The reported ease of division of labour between the development teams was highly valued by the participants. It seemed to be a natural continuation of the work division which existed before commercialization. Also, some of the scientist-developers who joined the projects were colleagues of the study participants, and hence it was clear from the beginning in which part of the development they would be interested and engaged. At the same time, participants admitted that many parts of the software were only understood by their original creators (if they were understood at all). Interestingly, the participants did not explicitly associate these two events.

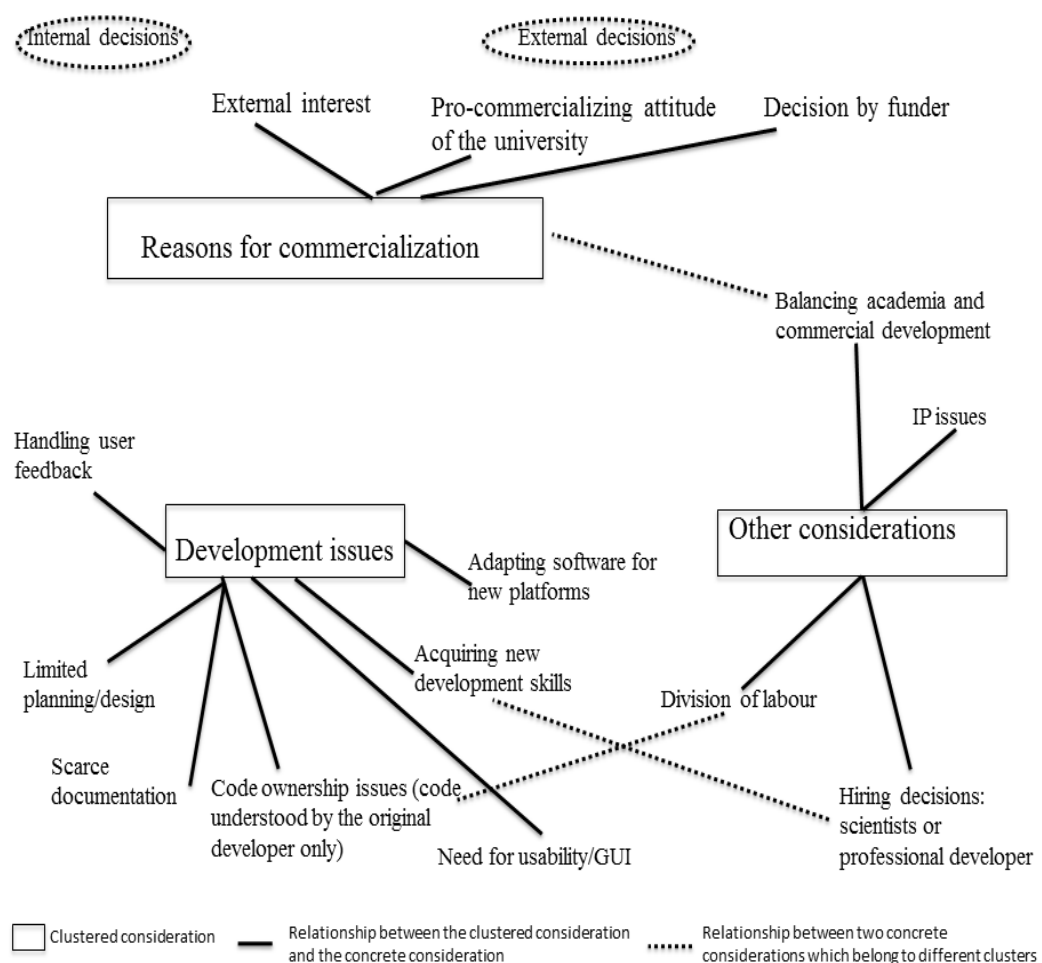
Despite all the challenges and effort which commercializing scientific software introduced, most participants still perceived themselves primarily as academic researchers rather than entrepreneurs. Commercialization of their software was certainly one of their major undertakings, yet it was seen as either another output of their research work or an activity embedded within the research practice. This hybrid professional role identity among scientists who engaged in commercial activities was also noted by Jain et al. (2009). This preliminary study is consistent with Jain et al.’s observation that the researchers who reshape their careers and engage in commercial activity without leaving academia tend to be more senior academics. The only interviewee who decided not to stay in academia was at a relatively early stage of his career in comparison to the other interviewees. It should be also added that he did not actually set up a company but joined one started by another participant. He was not then a scientist-entrepreneur but rather a scientist-turned-engineer, as he described himself.

2.4.4 Conceptual findings from the preliminary study

The exploratory nature of this preliminary study meant that the concepts which emerged from it had a very wide scope. The figure below summarises the conceptual findings and shows how they are related. The decisions about the commercialization were either internal (so the scientist-developer decided to commercialize the software) or external (the decision was made *for* the scientist-developer). The diagram shows how the different reasons for commercialization can be categorised according to whether they were related to internal or external decisions.

The considerations related to “Balancing academia and commercial development” are related to the reasons for commercialization. The reasons which were more external could potentially mean that combining the roles of an academic and an entrepreneur might turn out more challenging than expected. “Division of labour” closely relates to the “issues with code ownership”. The decisions whether to hire scientists or professional developer could potentially have a direct effect on the need to acquire new development skills.

Figure 2.1: Conceptual findings from the preliminary study.



2.5 Defining the focus for subsequent research

This exploratory study revealed a range of interesting topics related to commercialization of software originally developed in the scientific end-user context. The two previous sections reported and discussed findings, ranging from career change and work organisation matters to concerns related directly to software development such as re-engineering the source code, learning and applying new skills, and software documentation. The topic of software documentation was discussed by every single interviewee, and it was clear from their accounts that documentation played a key role in the transition from scientific end-user to commercial development. The lack of or incomplete documentation resulted in major problems with the software, and it was probable that many of these problems would persist, because they were expensive to address.

This outcome of the study raised further questions about the role of documentation in scientific software development. How much is enough when it comes to documentation? What information is essential to be captured and why? What are the forms in which information about the software and/or the software process is captured in scientific software development? Is the role of documentation the same in different contexts of scientific software development? The relevant literature did not answer these questions. The literature review presented in Chapter 3 shows clearly that none of the published studies focused on documentation in scientific software development. A few researchers reported some information about documentation, but this was not sufficient to address the above-mentioned questions. On the other hand, there were several studies on documentation in commercial or open source (but not scientific) software development. In order to fill in this gap, the main focus for subsequent research became documentation in scientific software development. The first main study, reported in chapter 5, focused on production of documentation by scientists-developers. The studies which followed were designed to respond to findings of the preceding study.

Chapter 3 Literature review

3.1 Introduction

The literature review covered in this chapter explains in more detail the different contexts of scientific software development and looks at the role of software documentation. The chapter provides a wider view of the latter, drawing upon selected publications related to documentation outside the scientific software development context, in order to understand what has been conceptualized so far about the subject. This allows for clarifying the conceptual contributions of this thesis. The review looks also at issues related to documentation writing (both in and beyond scientific software), which will later help in understanding the practical application of some outcomes of this thesis. The review of the selected publications on communities and networks of practice is useful in understanding the role of the community in scientific software development.

This chapter starts with an overview of different contexts in which scientific software is developed. Then the characteristics of scientific end-user development are discussed, including a look at key topics such as: the transition from end-user development to developing for others; tools and programming languages for end-user development, and the particularities of end-user development in scientific software. The next section discusses recommendations for scientific software development practices such as documentation and testing. The next three sections focus specifically on software documentation: documentation in scientific software development; documentation in professional software development; the role of documentation in software maintenance. A discussion on reasoning and assumptions made about the knowledge of those who use and read documentation is presented. Finally, the role of communities of practice in scientific software development is considered.

3.2 Contexts of scientific software development

It is possible to distinguish at least three contexts of scientific software development, based on who the developers of scientific software are. In the first context, scientific software is developed by professional software developers who typically do not have a background in science. In the second context, scientists and professional developers write source code together. In the third context, scientists develop software on their own.

This third context also includes a common situation in which the scientist-developer is also the only user of the given piece of software. Often referred to as “scientific end-user development”, this is relatively common in scientific software development (Segal, 2008b); almost all of the participants of the empirical studies presented in this thesis started as scientific end-user developers. Therefore the context of scientific end-user development is covered separately from the point of view of end-user development, in section 3.3, as well as in section 3.2.3 below.

3.2.1 Professional software developers developing for scientists

In this context, professional software developers develop software for scientists who participate in the development as users. The software may be written from scratch (for example, De Roure and Goble 2009 or Thew et al. 2009) or may build on a collection of tools that scientists are already trying to use (for example, Vidger et al. 2008). Scientists do not develop any software but actively participate in the development process as users. Such collaborations require willingness from both sides to build trust and mutual engagement (De Roure and Goble, 2009). It is not clear whether any of the expertise or previous experience in software development that the scientists may have affects the collaboration in any way. Potentially, by working together with professional software developers, scientists may be able to pick up some good practices related to software development, including documentation writing.

De Roure and Goble (2009) emphasize the importance of keeping a degree of flexibility and being ready to accept the changing needs of the scientific users. The authors also mention that committing some of the development time to working on scientists’ problems which were not directly related to the developed software benefited the professional software developers, who thereby had an opportunity to understand the context of the scientists’ work. Effective collaboration provides an opportunity to learn about the others perspective.

Segal (2005b) discusses the differences in approach to requirements gathering and documenting between the professional developers and the scientists. In her study, the professional developers expected the user requirements to be well defined and captured in a formal document. At the same time the scientists did not understand the role of such document, as, in their experience, requirements were changing all the time. The scientists noted the practice of documenting requirements but were reluctant to pick it up. Writing formal requirements documents may in fact not be suitable in scientific software development. However, it is not clear whether capturing some of the initial ideas related to software development would be useful for scientists. The published literature does not investigate if scientists do informal requirements documentation.

In contrast to formal requirements documentation, the informal approach emphasizes that software methodologies should be tailored for the needs of scientific software development. For example, the necessity of accommodating emerging requirements is also mentioned in (Thew et al., 2009). In this case, of software supporting research in epidemiology, the requirements were elicited using a number of techniques including unstructured interviews, user observations, domain knowledge workshops, and scenarios. The professional software developers were also open to the fact that the requirements may change and that the software for scientists will be developed in small steps. This iterative process of development resonates with the model of scientific software development proposed by Segal (2008b), which assumes that the requirements may be revised and changed in every iteration step. Ackroyd et al. (2008) emphasize the importance of close and constant communication when developing for scientists. Establishing collaboration with scientists, treating them as partners in the project, and involving them in the work rather than keeping them away as much as possible is key to successful scientific software development.

3.2.2 Scientists supported by professional software developers

In this context, the scientist-developers and the professional developers work in one team developing the software, which typically supports the research performed by the former. The division of tasks may vary, but the scientists are more than just passive users. They actively contribute to the source code and to other activities related directly to software development.

Such collaborations between professional software developers and scientist-developers may be challenging. In her study of the development of a laboratory information management system Segal (2009) found an example in which the collaboration between the professional software developers and one group of the scientist-developers (also called “professional end-user developers”) led to a conflict and eventually had to cease. The reason behind the conflict was that there were differences in values ascribed by the scientist-developers and the project manager (who was a professional developer) to the skills and knowledge related to software development. The scientist-developers perceived research and the skills necessary to do it as highly important and more valuable than software development skills. Despite the scientists having first-hand experience of developing software, research remained the top priority for them.

In cases in which the scientific end-user developers contribute less to the coding itself, similar conflicts between them and the professional software developers may still emerge (Killcoyne and Boyle, 2009). The authors note that “the common attitude toward software within research is that it’s not “real work” even if professional engineers are involved” (Killcoyne and Boyle, 2009, p. 21). Inevitably, the perception of software development as almost a side activity (not real work) means that fewer resources

and less attention are allocated to it. It is possible that, in due course, the scientists realize that potentially such an attitude may result in problems with scientific software maintenance, robustness and correctness.

3.2.3 Scientists as developers

In this context scientists are the only developers, either working alone on the software or together with other scientists but without the involvement of professional software developers. The software is often developed in the scientific end-user context. That is, it is intended for only the scientist-developer's own use or for use in his local laboratory (Segal, 2007). However, sometimes it happens that the software developed in this context is also used for teaching, or that it moves out of the original research environment and is applied in other related areas (Sanders and Kelly, 2008).

In the context when scientists develop software on their own, typically their primary goal remains advancing their research and the given software is only a means to an end (Segal, 2008b). Scientists focus on different goals than professional software developers. For example, in High Performance Computing, aims which seem apparent to professional software developers, such as improving program performance, have a lower priority for scientist-developers who focus on "executing science" (Basili et al., 2008). This resonates with the observations discussed earlier in this chapter that the software development practices of scientist-developers differ from those of professional developers largely not because scientists are not aware of these practices but because they do not fit the context and priorities of their everyday work.

As Basili et al. (2008) points out that transferring any existing software engineering practices to the High Performance Computing community where possible requires tailoring them for this particular domain. Successful adoptions of programming methodologies, such as Extreme Programming, can be possible (Wood and Kleb, 2003), but good understanding of the context is crucial. This is also echoed by Easterbrook and Johns (2009), who conducted a case study of software development for climate models. The authors observed that the scientist-developers used several software engineering methodologies and tools (such as PRINCE project management technology and version control). The adoption of software engineering practices and tools followed one general rule: if they matched the scientists' needs, they were used in the project.

3.2.3.1 Scientist-developers in High Performance Computing

Practices related to scientific computing within High Performance Computing (HPC) tend to be treated separately within the literature. HPC is typically used in computationally-intensive domains in which

research requires implementation of advanced and complex models and algorithms. The knowledge needed to understand and apply these models and algorithms is usually so advanced that scientific software is developed by experts in the domain, the scientists themselves (Basili et al. 2008). Not only is High Performance Computing important in computational science because it is widely used by scientists, but HPC instrumentation may also potentially increase research competitiveness (Apon et al., 2010). Even though the study conducted by Apon et al. (2010) shows that there are a number of other factors than just HPC instrumentation that may have a positive impact on the increase of research competitiveness, the findings indicate that HPC is important as well.

In their study of different scientific HPC projects, Basili et al. (2008) note some differences between the practices of the scientific HPC community and the “traditional SE [software engineering]” (p. 29). The authors note that “the goal of scientists is to do science, not execute software” (Basili et al., 2008, p. 31) which observation is consistent with the findings on other than HPC contexts of scientific software development (for example, Segal 2008b). In the case of HPC codes, that means that scientists rarely optimize their code for better performance, which in generic HPC software development (for example, as done by computer scientists) is a common practice. And if scientists decide to optimize their code, it often involves reworking in depth the underlying computational model. In addition to that, maintainability and portability of the codes are highly valued, and, if improving the performance may potentially result in affecting maintainability and portability, scientists are reluctant to make such improvements. Validation and debugging of HPC scientific software codes are typically challenging in many domains due to the lack of an oracle with which to compare the results. Basili et al. (2008) recognize that some of the practices widely adopted among software engineers do not fit scientific High Performance Computing. However, they indicate that adopting frameworks may improve scientists’ productivity: “Scientists have yet to be convinced that reusing existing frameworks will save them more effort than building their own from scratch.” (Basili et al., 2008, p. 34).

It should be noted that the matter of HPC programming productivity has been the focus of one of the large-scale research projects funded by the Defense Advanced Research Projects Agency¹ (DARPA). Halverson et al. (2008) report their experiences from conducting “an ecologically valid study of programmer behaviour for scientific computing” (p. 1). The authors’ account shows the extent of the effort that needs to be put into this kind of empirical study, and at the same time it provides evidence that this investment of effort and overcoming challenges is necessary to understand practices related to scientific software development.

One of the main challenges in HPC development is the increasing complexity - successful HPC devel-

¹<http://www.darpa.mil>

opment requires a set of skills in science, programming, scaling, and management (Squires et al., 2006). Individuals who have this complete set of skills are rare. According to the authors, the practice of building and expanding teams of scientists and programmers to address the expertise gap is not a complete solution to the complexity challenge. Squires et al. (2006) argue that productivity can be improved by “transformation in the way HPC applications are developed and maintained” (p. 9).

The need for changing the HPC practice in scientific computing is echoed in another study by Faulk et al. (2009). First, the authors note the differences in practices and tools related to software development between computational scientists and software engineers. For example, they note that “current software engineering practice intentionally abstracts away from (...) hardware properties [performance, hardware costs and portability]” whilst “issues important to software engineers (highly maintainable code, robust programming languages and practices, increasingly higher abstraction levels, time to market) receive almost no attention from scientific programmers” (Faulk et al., 2009, p. 32). Second, based on these observations, the authors argue that the practices popular among software engineers are often not applicable in the scientific software development context. Three strategies are identified as potentially useful for computational science: automation, abstraction, and measurement. A set of initial experiments in rewriting the code led to a conclusion that: “Success will require revisiting many common assumptions in software engineering and then re-engineering those solutions accordingly. It will also require far greater communication and collaboration between the software engineering and scientific computing communities.” (Faulk et al., 2009, p. 38).

The literature related to scientific software development in the HPC context does not investigate in depth the matters of documentation. However, as many of the participants of the empirical studies reported in this thesis had worked in projects done on supercomputers, the review in this section served understanding the particularities of the work and software development practices within HPC.

3.3 Scientific end-user development

Many scientists start their experiences with software development as end-user developers (see for example, Segal (2007) or Carver (2007)). There is a growing recognition that scientists may constitute a particular group of end-user developers (see for example, the special issue of *Journal of Organizational and End User Computing (JOEUC)*, 23, 2011). This section looks at some general characteristics of end-user development and then discusses the literature focusing particularly on scientific end-user development.

3.3.1 End-user development - definition

According to Ko et al. (2011), the difference between end-user developers and professional developers is based on their goals: “professionals are paid to ship and maintain software over time; end users (...) write programs to support some goal in their own domain of expertise” (Ko et al., 2011, p. 21:2). Professional developers may not be the users of the software they develop. And the software that they work on *is* the ultimate goal. For end-user developers, software is a means to an end. The definition of end-user development is closely tied up with how software development, or more generally, programming is defined. As one of the first researchers who investigated end-user development puts it: “[for end-user developers] The objective of programming is to create an application that serves some function for the user. From the end user’s point of view, the particular behaviour involved is not important, so long as application development is easy and relatively rapid.” (Nardi, 1993, p. 6). Ko et al. (2011) define programming as “the process of planning and writing the program” and a program as “a collection of specifications that may take variable inputs and that can be executed (or interpreted) by a device with computational activities” [p. 21:4]. Fischer et al. (2004) state that “end-user development activities range from customization to component configuration and programming” [p. 33].

The secondary role of software development that serves another goal means that the activities related to the development are given lower priority. Investing time and effort in software development and related activities may remain beyond the scope of interest of end-user developers.

3.3.2 From end-user developers to developing for others

Ko et al. (2011) note that the same person may be a professional developer and an end-user developer. It all depends on the context. For example, someone working as a professional programmer may develop a short script to help him set up and configure his machine to suit his needs when he is working. The script helps him to achieve the goal (configure his machine and thus work more effectively) but it is not the goal itself.

Source code developed by end-user developers is typically intended for their own use (Burnett et al., 2004). But there may be cases in which an end-user developer undergoes a change and becomes a developer who creates software for other users (Ko et al., 2011). Then the aim of an (originally) end-user developer may shift toward producing the software for a potentially wide group of users, including the end-user developer himself (Ko et al., 2011). The simplest example may be someone who created a website template for his own use, to set up a personal website. It turns out that another (for example, a friend) happens to like the template and requests the template developer for a copy of the source code. This may potentially evolve into a situation in which the original developer starts developing more templates and sometimes even sells them. The question arises whether the software development practices

he acquired as an end-user developer change or remain the same when the context of the development changes.

3.3.3 Tools and programming languages for end-user development

There have been a number of end-user development languages and tools developed to accommodate end-user developers (Ko et al., 2011). For end-user developers, the choice of the programming language may depend primarily on their accessibility, ease of use, and ability to provide quick solutions. Fischer et al. (2004) indicate that “a key trade-off in end-user development languages is between their scope of application and learning costs” [p. 34]. A high (broad) scope and cost of learning are associated with generic programming languages such as Java which can potentially be used to write most types of software. Domain specific languages often have low costs of learning, but at the same time their scope of application can be very limited.

The choices which end-user developers make when it comes to software development may be mainly pragmatic. The tools and languages which are considered to make the development easier and faster may be preferred over other tools and languages which may, from a professional software development point of view, be a better fit for a particular problem.

3.3.4 End-user development and scientific software

In scientific software, development is secondary to achieving the main goal which is progressing one’s research Segal (2007). Software is very often developed and used by exactly the same person - a scientist who needs to address a particular research problem. The main scientists’ interest lies in the researched problem, and, once the software is good enough to help to address it, the scientists leave it aside (Segal and Morris, 2011).

For many scientific end-user developers learning a general purpose programming language used widely in professional software development does not pose major challenges (Segal, 2005a). As Segal notes, scientists, trained and used to manipulating formal languages and abstract models, find it relatively easy to learn programming languages and concepts. It should be however noted that not all scientists feel entirely comfortable with using programming languages and may not be eager to program themselves (Chilana et al., 2011). There are custom-made tools created specifically to support end-user programming for scientists (Letondal, 2006). And even if the ways in which non-programmers use these tools may not meet the expectations that computer scientists have, as long as the goal is achieved the elegance of the program should not matter (Begel, 2007). In fact, as Begel (2007) states, “[computer scientists’] goal is to enable all scientists, even those who are apprehensive about computer programming, to create

and study their own models of complex systems and use them in their research” (p. 2).

Scientists, in particular in disciplines with a mathematical background, are likely to find programming per se relatively easy (Killcoyne and Boyle, 2009). Yet, “there’s a lot more to software development than merely producing some software that appears fit for purpose.” (Segal, 2005a, p. 38). Popular textbooks in software engineering (for example, Sommerville 2004) discuss a whole range of aspects and recommendations for the practices of software development apart from learning a programming language syntax. And as Segal further notes, “There’s testing to confirm that the software is, indeed, fit for purpose; there’s constructing the code so that it is comprehensible for future maintainers (...), and there are issues to do with increasing productivity and dependability” (Segal, 2005a, p. 38).

As noted earlier, applying software engineering methods in scientific software development may not work. This is also true in approaching the scientific end-user development (Segal, 2009).

3.4 Recommendations for scientific software development practices

Changing scientific software practices may be challenging. Concluding on two workshops about scientific software, Kelly et al. (2011a) emphasize that the processes and tools developed by software engineers to manage the complexity of software in general may be difficult to adopt in scientific software context. Peer code review “entails the large overhead”. Scientific software design is difficult due to the transitions of the expressions of theoretical concepts into their coded forms. The software development models such as waterfall, which often works well in industrial software development, “doesn’t fit well with the realities of scientific software development” (Kelly et al., 2011a, p. 8).

The literature discussed in this section indicates that there are different contexts in which scientific software is developed. The practices related to scientific software development differ within these contexts. These practices often differ from the practices used by software engineers who do not deal with scientific software. As Kelly (2007) points out in her position paper, a chasm has developed between the scientific-community and the software engineering community. Glass (2004) referenced by Kelly states that: “We need a mapping (...) between the methodologies that software engineers are developing and the problems that application-domain people are solving.” Merali (2010) highlights that scientists’ practices related to software development may result in some serious issues (erroneous software, issues with its maintenance and so on). The author notes some solutions for addressing these issues. One of these solutions is promoting openness and requiring scientists to publish their source code and the data. Another solution is creating nationwide or even international initiatives to provide support to scientists writing code. A long term solution, according to the author, is providing relevant training to graduate

science students.

3.4.1 Recommendations for documentation

In addition to the literature reporting empirical studies on the practices related to scientific software development and experience reports, several authors publish recommendations and advice for developing scientific software. A brief overview of a selection of these publications adds to the understanding of the state of practices related to scientific software development. O’Leary (2006) provides advice on working with scientific legacy code. What is interesting in the context of this thesis is the guidance on documentation. O’Leary (2006, p. 78) lists the following information to be included in order to “help a potential user decide whether the software is on interest:

- the code’s purpose;
- item the author’s name;
- the date of the original code and a list of later modifications;
- a description of each input parameter;
- a brief description of the method and references.”

The author also suggests documenting the test code, which should exercise every statement in the target code as well as be archived. In addition, “The testing code should compare against some trusted result” (O’Leary, 2006, p. 80). However, sometimes obtaining these trusted results may be a challenge, as “the amount of information available for a set of oracles is limited” (Kelly and Sanders, 2008, p. 7)

3.4.2 Recommendations for testing

Kelly et al. (2011b) try out in an exercise analysing scientific software testing using four dimensions of testing: context, goals, techniques, and adequacy. The understanding of the context is needed to establish what matters for the test. Understanding the goals helps in “refining techniques and adequacy criteria to match them” (Kelly et al., 2011b, p. 89). The technique needs to be chosen after considering the goals as well as the tester’s knowledge and skills. The adequacy criteria need to “focus on both the product and the scientist”. The exercise carried out by the authors show that indeed scientific software development characteristics mean that approaches commonly used by software engineers may not work for scientific software.

In another paper Kelly et al. (2009) recommend “five practices for computational scientists who write software”. The first one is separation. The code implementing the scientific model is separated from the source code of the user interface and the data preparation and output is separated from its manipulation. The second practice is organisation. The key is the mindset of a particular scientist, but using tools

such as a version control can be a great support. The third practice is review. Scientists are used to publishing and reviewing their scientific models but rarely do so with their code. However, code inspections can be very beneficial and help eliminate errors which, if unnoticed, may propagate and even lead to incorrect results. The fourth practice is testing. The authors recommend to “test the code and the science, but test the code beyond the science” (Kelly et al., 2009, p. 51). And the final practice is simplification. The authors discuss several strategies including breaking the application into small subprograms and parallelizing and reusing existing and well-understood libraries. The authors conclude that their suggestions should not be treated as silver bullets, but are backed up by successful examples.

3.5 Documentation in scientific software development

This section discusses the information about documentation production and use reported in scientific software development and identifies the gaps in the literature in this topic. Documentation production in the scientific software development context has not received much attention from researchers to date. The published empirical studies of scientific software development focus on other selected aspects and practices, for example on collaboration (Segal, 2009) and testing (Hook and Kelly, 2009), or explores a broader picture of scientific software development practices, within a particular subset of scientists using HPC (Basili et al., 2008) and across various disciplines (Sanders and Kelly, 2008).

3.5.1 Reasons for not writing documentation

User documentation is usually not produced by scientists developing scientific software, unless there is anticipation that the software will be used outside the scientists’ research group (Sanders and Kelly, 2008). In fact, documents other than user documentation are almost never produced in the scientific end-user context (Segal, 2007). However, the findings from the survey conducted by Nguyen-Hoan et al. (2010) indicated that some scientists may write user manuals even if software is anticipated to be used by themselves or a group of users local to them. It is unclear whether this kind of user documentation has any particular characteristics and differs from user documentation produced for software which is meant to be used by a wide community of scientists. In general, if the user base is small (or perceived to be small), it usually is a reason for not producing documentation (Nguyen-Hoan et al., 2010).

One of the reasons for not producing documentation reported by Nguyen-Hoan et al. (2010) was the need to commit a significant amount of time and effort to writing documentation. Documentation may also not be produced due to constantly changing requirements. Scientific software is typically developed within research projects, and these are likely to “break new ground”, explore and discover new areas (Aranda et al., 2008). In addition, the concept of a requirement document may appear foreign to scientists (Segal, 2005b). Scientists are used to writing scientific papers. Such publications aim to

present scientific results and argue that particular conclusions are correct. Requirements documentation is a kind of contractual agreement. It states the needs or the way these needs will be addressed. It uses different concepts, structure and vocabulary than those typically used in scientific publications. For professional software developers, the production of such document is often an expected part of software development, whilst for scientists it is something they do not perceive as necessary (Segal, 2005b).

3.5.2 Issues with scientific software documentation

Lack of or poor documentation in scientific software projects may lead to the situation in which knowledge about the software is shared in a different way than via documents. The community of practice may be perceived as more effective in knowledge sharing and may replace documentation (Segal, 2007). However, if such a community is not very stable, this approach to knowledge sharing may be problematic. For example, if in a scientific software development project the turnover of the team members is high, knowledge leaves with those who leave the project, making it difficult for newcomers to get up to speed. Some scientific software teams may develop their own documentation regime to share understanding about the software (Easterbrook and Johns, 2009). In particular the teams may use electronic media to enhance their communication.

3.5.3 Scientific software documentation stakeholders

Stakeholders of scientific software documentation may be divided into groups based on the purpose for which they need documentation. This is directly connected to their relationship with the software, as developers, users, or both.

Scientific end-user developers may not need user manuals (Segal, 2007). Since the same person is the developer and the user, directions for using the software are not essential. The software development and use cycle may be short enough for them to rely only on their memory. Further, constantly emerging requirements (Carver et al., 2007) result in frequent changes to the software and if there was software documentation which would have to be kept up to date this would potentially be very costly in terms of resources. However, Nguyen-Hoan et al. (2010) report in their survey results that user manuals may be written, even if software does not go beyond the scientific end-user context. This raises a question whether different groups of end-user developers may have different needs when it comes to documentation.

The users of scientific software who are not its developers are provided with instructions on how to run the software (Sanders and Kelly, 2008). However, these users are typically those from outside of the immediate scientist-developers' environment. It is possible that, when it comes to writing manuals,

users are divided into two groups. One group is users who work within the same research team as the scientist who developed the software. The other group is users outside of that team. The former can have direct access to the original developers. A written manual may be essential for them.

Another stakeholder group may be professional software developers who develop scientific software. These “domain-independent software engineers” (Killcoyne and Boyle, 2009, p. 21) are typically used to working in small teams in which the exchange of information about software happens often and dynamically. Whilst in research projects, they may end up being isolated in their tasks, as “The discovery-oriented, project-focused, publish-led culture of research encourages scientists to work in small isolated silos.” (Killcoyne and Boyle, 2009, p. 21). It is then common that the professional developers become a one-man band covering all aspects of software development, including capturing and distributing information about the software. Nevertheless, professional software developers who work with scientists on scientific software development may have different expectations about documentation than the scientists with whom they collaborate (Segal, 2005b).

3.6 Documentation in professional software development

This section of the literature review focuses on selected publications reporting the results of empirical studies of documentation practices of professional software developers. As discussed earlier, the priorities and work cultures of scientists and professional software engineers may be very different (Segal, 2005b). As a result, collaboration between scientists-developers and professional software engineers may lead to challenges that compromise the completion of the projects in which they are engaged (Segal, 2009). The question is whether their practices related specifically to documentation are different too. More specifically, how do professional software developers actually produce and use software documentation? Knowing more about the documentation practices of professional software developers may provide additional insights into investigating the documentation practices of scientists-developers.

3.6.1 Documentation practices among professional software developers

In professional software engineering, the practices of production and use of documentation may vary from the recommended processes (Lethbridge et al., 2003). Based on a set of empirical studies (interviews, a survey, shadowing participants at their workplace, and “companywide tool use statistics”), Lethbridge et al. (2003, p. 37) discuss software documentation practices of software engineers who work in industry. Most of the documentation tends to be out of date. Software engineers typically update the documentation a few weeks after the changes to the code are applied. The bigger the changes to the code, the bigger differences there are between the information in documentation and the state of the software. The exception was testing and quality documentation, which was reported in the survey by

almost 40% of 45 respondents to be updated every few days. This was also the type of documentation most often consulted. Interestingly, the comparison of the findings from the survey and interviews to those from the observations revealed a discrepancy between what the software engineers said and what they did. In general, the participants said that they consult documentation more often than was observed by the researchers. Overall, the software engineers preferred to both produce and use “simple yet powerful” documentation rather than “complex and time-consuming” documentation. They valued and kept up to date comments in the source code, as these were “short and right there”. The researchers suggested that software engineers “consciously or subconsciously” evaluate which documentation needs updating.

3.6.2 Documentation in open-source projects

The main reason for considering documentation in open-source projects in a separate section of this literature review is to understand what could be the role of a potentially very wide community in writing documentation. Scientific software, developed within a research (academic) environment, may not explicitly be labelled as “open-source” (with explicitly stated license and a clear contribution guidance and workflow) but, in fact, it often is. Scientists share their source code, pick up, extend, and maintain software developed by someone else. In particular, this section provides background for the final empirical study (reported in Chapter 7) which examined a case of open-source scientific software with a considerably large developer and user community.

3.6.2.1 User-driven documentation in open-source development

Berglund and Priestley (2001) propose user-driven documentation production taking advantage of open-source projects. The authors suggest an open-source documentation framework. According to them, the following conditions must be met before one can even think about user-driven documentation: electronic format of the documentation, accessible and manageable via a website service, open source license (for example GNU Free Documentation License¹), a possibility to split the documentation for branching out projects, and at least a first prototype of the software package in question. Then the authors suggest that the following elements of the infrastructure should be established: a relevant social structure among the community should be established, goals and milestones laid out, infrastructure for live communication and managing the work, and so on. Another recommendation is for developing a documentation life-cycle and nominating editors from among the community. Finally, the authors conclude that “success ultimately depends on the open-source documentation project’s ability to accumulate enough users that can and will contribute to the process” (Berglund and Priestley, 2001, p. 140). Indeed these recommendations and the conclusion seem reasonable and likely; however, a relevant empirical study could really show whether “it would work” as Berglund and Priestley (2001) put it. Study 3 (as reported in

¹<http://www.gnu.org/copyleft/fdl.html>

chapter 7), which investigated documentation crowdsourcing open-source scientific software documentation, partially fills in this gap.

3.6.2.2 Factors influencing documentation in open-source development

A study looking at documentation practices in the context of six different decision points in open source software development revealed some interesting aspects of documentation production (Dagenais and Robillard, 2010). The participants of this study had at least three years of experience in software development, but it is unclear if all of them were *professional software developers* as it is defined for the purpose of this thesis. At the initial stage of the project, the first two decisions are made: the first about selecting the tools supporting documentation production, and the second about the type of documentation which needs to be created. Then, during the stage of “incremental changes”, decisions are made about: adapting the documentation to project evolution and managing documentation contributions from the community. In stages called “bursts” (major concerted changes), the contributors to open source projects may, for example, rewrite the whole documentation because they are asked to write a book about their project. Another decision which may be made at this stage is changing the documentation infrastructure. The study also showed that engaging a wide community in creating documentation for an open-source project through a wiki poses several challenges. First, a wiki lacked authoritative-ness - there was no guarantee that the information included there would be up to date and correct. Also, since modifying documentation did not require much effort, there was a threat that it would become “less concise and focused over time”. On the other hand, the perception was that, since the community was less inclined to produce documentation than to produce the source code, the entry level to write documentation needed to be lower than to write the code. A wiki infrastructure offered such a low entry option. At the same time, assigning documentation tasks to a specifically-selected team posed other problems. As the documentation teams were out-numbered by the contributors to the source code, they could not keep up with the changes in the source code, and hence the documentation lagged behind the software.

This brief overview of the literature on documentation practices of professional software developers shows that documentation use and production in real-life settings may not adhere those recommended (Lethbridge et al., 2003). Considering particular contexts of software development, such as open source development, may bring some new insights into software documentation production and use. There is a need for “the empirical investigation of human processes in realistic contexts and settings” to answer a set of questions regarding software documentation (Briand, 2003, p. 2). Even though Briand’s deliberations focus on analysis and design documentation in object-oriented development and then on

documentation in eXtreme Programming (XP), his statement can be extended to other cases of software development. Briand emphasizes the importance of empirical evidence and states explicitly that, in cases in which the evidence is missing and hence the expert opinions prevail, “their [the experts’] opinions often (...) conveniently match the author’s viewpoints or commercial interest”(Briand, 2003, p. 2).

3.7 Documentation and software maintenance

The software lifecycle rarely finishes at the moment of the softwares release. Once it is released, it typically requires maintenance, which may become as effort- and time-consuming a task as the initial software development. Software maintainers may need to look into different parts of the source code, and extend, adjust, or fix it depending on the circumstances. These activities require information and knowledge about the software, which one expects to find in software documentation. Documentation plays important role in software maintenance (Das et al., 2007) and, as the outcomes of the preliminary study presented in the previous chapter suggest, it may be particularly crucial in the case of scientific software.

3.7.1 What is software maintenance

Software maintenance, even if seemingly involving similar activities, differs from software development. Software development focuses on developing and delivering a product, whereas software maintenance can be treated as a provision of a service (Niessink and Van Vliet, 2000). Products are tangible and relatively (in comparison to services) less mutable. Services can change more dynamically responding to customers or users needs, but at the same time “cannot be saved, stored (...), returned or resold” (Niessink and Van Vliet, 2000, p.105).

According to the IEEE International Standard - ISO/IEC 14764, *software maintenance* is “the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage. (...) Pre-delivery activities include planning for post-delivery operations, supportability, and logistics determination. Post-delivery activities include software modification, training, and operating a help desk.” (p. 4). There can be different types of maintenance and the standard lists the following:

- **“adaptive maintenance:** the modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment. Adaptive maintenance provides enhancements necessary to accommodate changes in the environment in which a software product must operate. These changes are those that must be made to keep pace with the

changing environment. For example, the operating system might be upgraded and some changes may be made to accommodate the new operating system.”(p.3)

- “**corrective maintenance**: the reactive modification of a software product performed after delivery to correct discovered problems. The modification repairs the software product to satisfy requirements” (p. 3)
- “**emergency maintenance**: an unscheduled modification performed to temporarily keep a system operational pending corrective maintenance. Emergency maintenance is a part of corrective maintenance” (p.3)
- “**perfective maintenance**: the modification of a software product after delivery to detect and correct latent faults in the software product before they are manifested as failures. Perfective maintenance provides enhancements for users, improvement of program documentation, and re-coding to improve software performance, maintainability, or other software attributes” (p. 4)
- “**preventive maintenance** the modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults” (p. 4)

As Niessink and Van Vliet (2000) note, maintenance as defined and categorised above is more of a service “aimed at keeping the system usable and valuable for the organisation” (p. 106). In the scientific context, software remains valuable as long as it can effectively support research activities. Evolving science calls for adaptive maintenance, though other types of maintenance can be present as well.

3.7.2 Documentation and software maintenance

Software maintenance may be affected by a number of different factors including: maintenance activity types, product features, peopleware, and process organisation (Kitchenham et al., 1999). One of the product’s features that has impact on its maintenance is the product’s quality - the higher the quality, the easier the maintenance. As Kitchenham et al. (1999) point out, one of the things which constitute the quality is the software documentation. It is particularly important in cases when the maintainers have little or no contact with the original developers. The authors point out that the documentation may be of different forms, and its completeness, accuracy, and readability should be assessed in order to understand the influence of the documentation on software maintenance. The authors also note that in the case of old software, “documentation is often poor or non-existent” (p. 372). While this may be true, the suggestion that the authors make next that “in such cases, maintenance engineers need specialized tools such as re-engineering tools” (p. 372) may not always apply. If documentation (in a formalized form) is missing, the maintainers may turn to other information resources.

de (de Souza et al., 2005) note ten published studies which consider, to a greater or lesser extent, the role of documents in software maintenance. The considered documents ranged from detailed requirement and specification documents to more high-level overviews of the system. The latter appeared to be of high importance for software maintenance. Yet, in some contexts of software development, including scientific software development, requirements are typically not documented at all (Segal, 2005b), and other formalised documents, such as software architecture documents, are likely not to be written either.

However, information about software does not reside solely within requirements or design documents. Software maintainers use a variety of resources and thus there are a number of factors which influence role of documentation in software maintenance. These topics are discussed in the following sections.

3.7.2.1 Source code and comments in code in software maintenance

Souza et al. (2005) conducted two surveys among software maintainers. The results showed that the source code itself and the comments in the code were considered the two most important documentation sources for those who maintained software. These two types of documentation were also reported as most often consulted. The next two types of documentation in terms of importance rating and usage were data models and description of the requirements. The higher level documentation types (such as architecture documentation) were not very important for the maintainers. The indication was that high-level documentation was usually used only once, in order to gain a general overview of the system. Therefore such higher level documentation was important but rarely used.

Source code may be used as documentation not because it is preferred but because there is nothing else that is available (Das et al., 2007). Source code tends to be considered as the most trusted even if it is not the easiest form of documentation to understand and follow. The source code can be consulted for corrective, adaptive, and perfective maintenance. That is, the maintainers will look into the source code when they need to fix a bug, but also when they want to add a new feature (Singer, 1998).

3.7.2.2 People as sources of information in software maintenance

Documentation is not the only source of information about software used by its maintainers. People are often rated highly as sources of information as well (Seaman, 2002). They may be “original developers of the system, writers of the original system requirements, current users and operators, other maintainers and experts” (Seaman, 2002, p. 6). Even though human sources are typically highly valued as “accurate” and “handy”, they are perceived as less valuable than the source code itself, perhaps because people’s memories may be perceived as failure-prone. It is also possible that some people describe the system they developed in subjective terms, and hence this information is perceived as slightly less trustworthy than the source code.

As Seaman (2002) found, different people had different availability to be accessed depending on their relationship to the software. The developers of the original release and the writers of original system requirements were rarely available for consultation. Typically the developers left the project or were busy with other development. The requirement writers were relatively more accessible, but it was still often not enough. For maintainers, access to these two groups was on their “wish list”. The in-depth knowledge about the software that the developers and the writers had was of great help for the maintainers. The users and the operators were more accessible but less frequently consulted. Usually they were asked for help when the maintainers wanted to “isolate a defect, design a fix or enhancement, or understand how the system is used” (Seaman, 2002, p. 6). There was no doubt that the easy access to users and operators was an advantage. However, the main problem was their lack of in-depth knowledge of the software design.

3.7.2.3 Documentation maintenance

It is not only the software that requires maintenance. Documentation (if it exists) needs maintenance as well. When software evolves, documentation must keep up. But software is often maintained using the “quick-fix approach” (Sousa and Moreira, 1998), resulting in further disconnection between the software and documentation, as whatever is done with the software is not reflected in documentation updates. Establishing and advocating software maintenance processes which would put strong emphasis on updating documentation together with any changes to the software may seem to be a solution. However, implementing such processes in organisations can be a challenge. In some circumstances, it may simply not work, especially if activities related to software development and maintenance are perceived as secondary to other goals, as in the context of scientific software development.

Organisations may recognize the issues with documentation maintenance, yet they will still do little to address the problems (Kajko-Mattsson, 2005). The effort required to make the necessary improvements is perceived as too much to invest. In addition, introducing a system which could help with documentation maintenance is typically not planned for.

Another problem is access to documentation. Documents may include the information that the maintainers need and seek, but they may either be unaware of the existence of such documents or they may not know where to find them (Das et al., 2007). The difficulty in tracking documentation adds to the general workload and means that the developers do not focus their attention on trying to retrieve the documents. Instead, they turn to people who may have the relevant knowledge about the software (as discussed in the previous section). The maintenance of documentation includes not only updating it

with the new information but also more meta-level tasks such as making sure that documentation can be located easily, is somehow categorized, and that this categorization corresponds with the changes in the software.

3.8 Reasoning about documentation readers' knowledge

Software documentation aims to communicate information about the software and the software development process to different groups of readers: the developer himself or herself, other developers, end-users, managers, and resource providers. Producing any documents inevitably involves making assumptions about the readers. This may seem a simple and straightforward process, but it poses a difficult question: how to estimate the readers' knowledge. That is, how to estimate what to communicate in the documents for a particular group of readers. This section presents most cited publications about the assumptions people make about others' knowledge.

Those who construct the message know the information they want to pass on to others. The estimation of others' knowledge may start with a model of one's own knowledge (Nickerson, 1999). Then, in iterative steps using information about the readers of the message, the model of others' knowledge is refined. The author of the message uses different clues to refine his or her model of the readers' knowledge. For example, the authors and readers may belong to the same working community, and this may provide clues to the author about what may be known or unknown to the readers. Belonging to a given profession or holding the same degree, may provide another set of clues. Refinement may also come from constructing implicit models of knowledge structures by inference: if a reader knows X, then he or she must know Y. Also, in some cases, if someone appears not to know something, it may be inferred that he or she will not know about certain other related things. Finally, refining others' knowledge model via levels of knowledge seems to be most relevant to scientific communities. As Nickerson (1999, p. 744) puts it, "people who understand concepts at a given level are also likely to understand related concepts at a less deep level, but it would not be expected that people who understand concepts at a given level would necessarily understand all related concepts at a deeper level".

However, the refinement of the model may be far from perfect, as in many cases people have a tendency to overestimate how much of what they know may be also known to others (Nickerson, 1999; Hayes and Bajzek, 2008). Individuals who have expertise in a given area may be even more likely to overestimate what others may know about the same topic. They may have a tendency to assume that certain things are common knowledge when they are not (Hayes and Bajzek, 2008). Even if the immediate feedback from the readers is available (for example, in spoken communication), the beliefs of the author about his or her readers' knowledge may still override the clues coming from the feedback (Fussell and Krauss,

1992). As de Jong and Lentz (2007) point out, professional writers have difficulties anticipating the problems which readers of functional documentation, that is the documentation which specifies what processes and functionalities the software supports, may experience. Since the responsibility for producing successful documentation relies on the professional writers, the need for empathy, in terms of authors attitude and cognition, is apparent. Attitudinal aspects of empathy are related to “communication experts and professional writers willing to consider the needs and preferences of their readers” (de Jong and Lentz, 2007, p. 2). Cognitive aspects of empathy are related to “communication experts and professional writers [being] capable of [considering the needs and preferences of their readers]” (de Jong and Lentz, 2007, p. 2).

In the study focusing on documentation production (as reported in chapter 5) the participants explicitly stated that they made assumptions about the knowledge that the potential users and developers of a given piece of software may have. Therefore the review of the selected literature discussing the mechanism of constructing assumptions about readers’ knowledge allows for relating the findings reported in the thesis to a more general context.

3.9 Communities of practice and scientific software development

This thesis focuses on studying work practices in real-life settings performed within a certain community. Therefore this section of the literature review presents the selected publications relevant to the topic of work practices in real-life settings performed within a certain community.

3.9.1 Communities and network of practice

As Wenger (2007) says, the practice may include:

“what is said and what is left unsaid; what is represented and what is assumed; (...) the language, the tools, documents, images, symbols, well-defined roles’ specified criteria, codified procedures (...) all implicit relations, tacit conventions, subtle cues, untold rules of thumb, recognizable intuitions, specific perceptions (...), underlying assumptions, and shared world views.” (Wenger, 2007, p. 47)

In the context of the work place, studying communities of practice rather than just analyzing the prescribed processes, may help to “reveal the tensions between the demands of process and the needs of practice” (Brown and Duguid, 2002, p. 95). Focusing on communities of practice enables a researcher to gain a better insight not only into how the particular tasks are performed but also how the context may influence the actions.

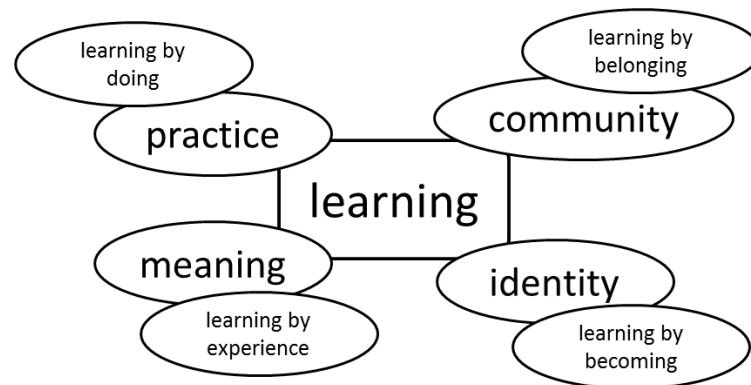
Work practice is almost always embedded in some kind of social setting (Wenger et al., 2002). The communities which exist within these social settings, by interacting with each other, develop common knowledge, understanding, and approaches which shape the practice. Communities of practice (Wenger et al., 2002; Wenger, 2007) can be of different size: scaling up from a few specialists to hundreds or thousands of people. Communities may be short-lived, for example, for the duration of a single project, or may last for several centuries. The members of communities of practice may be co-located or may work in a distributed environment. The latter has become even easier nowadays, in the Internet era. Communities of practice may be heterogeneous or homogenous - involving people from the same narrow discipline or from very different backgrounds. Communities of practice can exist within the same organisation or across different institutions. They can be formed intentionally or emerge spontaneously. Finally, communities of practice may be institutionalized or may remain unrecognized. That is, it is possible for a community of practice to exist and provide its members with various benefits but at the same time it may be completely invisible even for those engaged in it.

Brown and Duguid (2002) distinguish between networks of practice and communities of practice. The concept of networks of practice is closely related to communities of practice. Essentially networks of practice contain communities of practice and assume that these communities as well as their members may be connected into networks and exchange experiences and knowledge not necessarily in a face-to-face manner. Electronic networks of practice which use the internet and/or mobile phones as media of communication are one of the most common types of networks of practice.

3.9.2 Knowledge exchange in communities of practice

Communities (and networks) of practice provide their members opportunities to learn: new skills, new techniques, new methods, new approaches and so on. As Wenger (2007) argues, learning is not a separate and isolated activity. Learning happens all the time; it may only have different periods of intensity. Wenger proposes a social theory of learning which combines four components “necessary to characterize social participations of learning and of knowing” (Fig.3.1): meaning, practice, community, and identity. All these components are related, interchangeable, and “mutually defining” (Wenger, 2007, p. 211).

Figure 3.1: Components of a social theory of learning: an initial inventory. Source: Wenger 2007, p. 5



Actively participating in the activities constructs learning by doing (practice). Experience (which involves more passive engagement) contributes to developing meaning. An individual's identity formation is learning by becoming, whilst being a part of a community helps an individual learn how his activities fit within the community.

The concept of communities of practice “integrates all of these components” but at the same time is intuitive enough to understand as it refers to experiences which are known and familiar. For individuals, “learning is an issue of engaging in and contribution to the practices of their communities” (Wenger, 2007, p. 7). For communities, “learning is an issue of refining their practice and ensuring new generations of members” (ibid.). Finally, for organisations, “learning is an issue of sustaining the interconnected communities of practice through which an organisation knows what it knows and thus becomes effective and valuable as an organisation” (ibid.). The key point is that it should be remembered that learning does not happen only in educational institutions but rather takes place in almost all activities undertaken in real life. In particular, a lot of learning happens in work-related activities. However, these learning processes may not be easy to identify and, hence, instead of being accommodated and reinforced, remain unnoticed.

3.9.3 Communities of practice and scientific software development

It has been noted that in scientific software development the development teams may form communities of practice (Segal, 2009, 2007). In a scientific software project the scientists-developers work in the same scientific domain. They share “the same goals and the same values and customary behaviours”. At least some of them may have worked together for a few years, often in co-located settings. Most importantly, these scientists developed “tacit knowledge of individual strengths and weaknesses and of the distribution of both domain expertise and software development knowledge among the members of

the team” (Segal, 2009, p. 593). Being a member of a community of practice may influence a scientist’s choice with regard to the programming language and other tools supporting software development (Jones and Scaffidi, 2011). On one hand this may be an advantage, as a scientist uses the existing pool of knowledge about software development. On the other hand, the “good enough” solutions (Segal, 2007) may be short-sighted. That is, they will be sufficient to address the problem quickly, but, if the software is long-lived (which is sometimes the case), the “good enough” solution not only stops being sufficient but may actually generate further problems (Jones and Scaffidi, 2011). Communities of practice in the scientific software development context may not be recognised explicitly. For example, scientists-developers may develop a set of strategies to maintain shared understanding and particular software validation and verification practices (Easterbrook and Johns, 2009). They are not explicitly perceived as a community of practice, but their activities and interactions still fit into the definition of a community of practice.

3.9.4 Communities of practice and scientific software adoption

Communities of practice play an important role when it comes to adopting software. Scientists are likely to use a piece of software that was described in a peer-reviewed article or simply recommended by a researcher that they consider respectful and trustworthy (Joppa et al., 2013). By relying on such recommendations, scientists mitigate risks related to choosing software that might be faulty or not relevant. The mechanism described by “diffusion innovation theory” (Rogers, 1962) shows that adoption of innovations largely depends on what the members of the community, in particular opinion leaders, say. But as Joppa et al. (2013) note, scientists choosing a piece of software based on the peers’ recommendations may in fact still not feel confident as they actually do not understand what the software exactly does. Joppa et al. (2013) suggest what should be done to address this issue. They advocate changes in university training for scientists. The training should be designed so that the graduates would have sufficient knowledge in “statistics, computational methods, mathematics and software engineering” to assess scientific software in their domain. Another recommendation is for the change of practice related to scientific publications. Joppa et al. (2013) suggest that the source code related to the publication needs to be not only made available but also peer-reviewed by skilled reviewers. However such changes in the practices of scientific publications may be challenging. In their empirical study, Savage and Vickers (2009) showed that, despite the requirement from the publisher to share the data on demand, only one out of ten contacted teams of authors eventually provided the data. Even though the datasets are not the same thing as source code, they are related enough for Savage and Vickers (2009) to show how difficult it may be to enforce the practice of sharing the scientific software code.

3.10 Summary

It is possible to distinguish at least three different contexts in which scientific software is developed:

1. domain-independent professional software developers developing software for scientists;
2. professional developers and scientists developing software (writing source code) together; and
3. scientists working on their own.

In the last context, it is common that the developers are also the users of the software. For that reason, the documentation in scientific end-user development is scarce. However, if there is any anticipation that the software will be used outside the research group in which software was originally developed, a manual will be produced. Two reasons that scientists report for not writing documentation are: the lack of resources; and that they do not see a purpose in documentation as the software tends to evolve almost constantly.

There are indications that practices around scientific software development tend to emerge from the communities and networks of practice. Recommendations for scientific software development practices, and in particular for documenting scientific software, are sparse. Yet it is likely that, just as in professional software development, documentation plays a key role. Poor documentation may be a hindrance to software maintenance. When written documentation is not available or is insufficient, software maintainers look for information among other people involved in the software project. However, access to them may be difficult or even impossible.

Chapter 4 Research design

4.1 Introduction

This section presents the research design for the thesis. It discusses the overall approach to the planning and execution of the three main studies. It should be noted that each study was conducted and analysed independently. Therefore the exact details of the sources of the data for each study are presented in the respective chapters whilst the approaches to and methods of collecting and analysing the data are discussed in this chapter. The qualitative approach to the data collection is discussed, with a particular focus on the exploratory angle. Then the methods used in the studies are considered: semi-structured interviews, work-based interviews and the methods used for the case study. The following methods for data analysis are discussed: inductive coding, thematic analysis and quantitative analysis applied to the quantitative data collected as a part of the case study.

The structure of this chapter is as follows. Section 4.2 discusses the research approach. The choice of qualitative approach is justified. Then four main philosophical stances are discussed with the relevance to this research. The first three are briefly described whilst the selected philosophical stance, interpretivism, is discussed in more depth. Section 4.3 describes the approaches to data collection for all three main studies: semi-structured interviews in study 1; work-based interviews in study 2 and a case study in study 3. The ethical considerations are then mentioned. Section 4.5 discusses data analysis. The first part focuses on describing the inductive coding used in all three main studies as well as in the preliminary study. Secondly, the quantitative analysis used in study 3 is briefly discussed. The final section 4.6 considers limitations of the research design and the selected methods of data collection and analysis as well as validity assessment.

4.2 Research approach

This section discusses the qualitative approach and the philosophical stance, interpretivism, adopted for the research reported in this thesis. A justification of selecting the particular approach and stance is given. A detailed discussion of how the research reported in this thesis relates to the seven principles for interpretive field research (Klein and Myers, 1999) is included.

4.2.1 Qualitative approach

The main aim of the research was to explore the topic of documentation production in the scientific software development context. Real-life practices of scientists who developed software were investigated. The goal was not only to look at current practices but also to acquire some insight into past work practice in the scientific end-user development context. The aim was to increase understanding about the process and behaviors as well as the choices made by the people involved. The table below depicts how the aims of the research needed to be satisfied with the appropriate research approach and type of collected data. The rationale is discussed in the following paragraph.

Research aim	Needed data and approach
Explore the research area	Rich and descriptive data Open-ended questions Data-driven research
Real life practice	Empirical study with practitioners Rich and descriptive data
Understanding processes	Rich and descriptive data Empirical study
Studying and understanding behaviour and choices	Empirical study Descriptive data (explanations and descriptions provided by participants)

Table 4.1: Research approach and type of collected data required to meet the research aims.

Following the guidance on matching the research methods with the research aims provided by Robson (2002) empirical study with practitioners using qualitative methods to collect the descriptive data was the best approach. Qualitative data consist of stories, narratives and explanations - all that was needed to address the research aims and questions. The exploratory character of the study required data-driven research. That is, as the new themes were emerging from the data, the further direction for the study could be defined and followed.

Although the overall approach was qualitative, the case study described in 7 provided some quantitative data which was used in order to compare it with the findings from the analysis of the qualitative data. In case study research collecting data from multiple resources and comparing the findings is a recommended approach (Runeson et al., 2012).

Additionally, as the literature review did not provide a comprehensive view of the documentation practices in scientific software development, it was difficult to design a set of narrowly

focused questions which could be answered via a survey or through, for example, a quantitative analysis of any electronic records such as a software repository. Therefore the studies reported in this thesis had an exploratory character and aimed to collect information on a broadly defined topic of documentation production in scientific software development. Again, qualitative data was the best choice to obtain the rich and descriptive information (Robson, 2002).

In the case of this PhD research, not all concepts had a clear definition (for example, the user community was not defined as a continuum from black-box users to white-box users) and were emerging from the data. The research questions and subquestions which underpinned all studies were all “How..” and “What..” questions requiring descriptive answers and information about the context of the described phenomenon. The qualitative approach gave the researcher more flexibility in investigating different areas of the context without being restricted to only a number of its features.

4.2.2 Philosophical stance

This section briefly discusses how the research reported in this thesis relates to four different philosophical stances: positivism, Critical Theory, pragmatism and constructivism (interpretivism) (Easterbrook et al., 2008). The following paragraph consider what is the accepted knowledge and evidence for this research. The justification for the adopted empirical approach is provided.

The first stance, positivism, “states that all knowledge must be based on logical inference from a set of basic observable facts”(Easterbrook et al., 2008, p. 291). The main assumption is that the observations made are objective and that there is one objective truth about the studied phenomenon (Robson, 2002). Positivism and post-positivism dominates in natural sciences as well as quantitative studies and sometimes case studies. Post-positivism recognizes that scientific reasoning and the common every-day life reasoning are in fact quite similar. Post-positivism believes that there is an independent of one’s thinking reality but at the same time acknowledges that researchers are biased by their experience and background. In the studies reported in this dissertation positivism is not suitable. Post-positivism, due to its assumption of the existence of one objective reality is also not suitable. These studies explore human behaviour, decisions and approaches. They do not seek to establish one objective truth about scientific

software documentation production and use. This PhD study provides evidence for the importance of the context in understanding the studied phenomenon. Also, as is discussed further, the approach adopted for this research acknowledges that the researcher's previous experience and knowledge influences data collection and interpretation.

The second stance, critical theory, "judges knowledge by its ability to free people from restrictive systems of thought" and "critical theorists argue that research is a political act" (Easterbrook et al., 2008, p. 291). The research presented in this thesis does not mean to empower any particular social group nor does it aim to advocate for any social or political structure or system. Critical Theory is irrelevant in the case of the work presented in this dissertation.

The third stance, pragmatism, judges the knowledge "by how useful it is for solving practical problems. (...) truth is whatever works at the time" (Easterbrook et al., 2008, p. 292). Also "pragmatism adopts an engineering approach to research - it values practical knowledge over abstract knowledge" (ibid.). Even though this research hopes to provide some useful practical information, delivering practical outcomes is not the goal of this dissertation. The goal is to explore and understand scientific software documentation production and use rather than propose or implement any engineering solutions.

The third stance, interpretivism (or constructivism), "rejects the idea that scientific knowledge can be separated from its human context" (Easterbrook et al., 2008, p. 291). This stance underpins the whole research reported in this thesis. Interpretive inquiry assumes that a researcher's "knowledge of reality is gained only through social constructions such as language, consciousness, shared meanings, documents, tools and other artifacts" (Klein and Myers, 1999, p. 69). An interpretive research aims to investigate the complex reasoning of humans about a particular situation or phenomenon. In interpretive inquiry no dependent and independent variables are predefined. The subject of the inquiry is understood through meanings which people assign to it (Klein and Myers, 1999). An interpretive approach was adopted for this research for the following reasons:

- all studies had primarily an exploratory character and aimed to look at documentation in scientific software development from as wide a variety of perspectives as possible;

- documentation production and use practices were investigated from different points of view - including scientist-developers and scientist-users;
- the phenomenon of documentation in scientific software was described, analysed and interpreted solely through the information and insights provided by the participants of the subsequent studies;
- the outcomes of a previous study informed the design of the following study allowing the researcher to be explicitly guided by the data.

Klein and Myers (1999) propose seven principles for interpretive field research based on hermeneutic principles which are helpful in evaluating and conducting interpretive research. Since the interpretive stance underpins this dissertation, the following paragraphs discuss in detail how these seven principles relate to the presented research.

The first and the most fundamental principle is that of the Hermeneutic Circle. It states that the understanding of a complex phenomenon is a process in which the interpretation moves between understanding the elements of the whole and the surrounding context and the understanding of the single elements. That is, understanding of the big picture informs the investigation of the single elements. In subsequent iterations of the hermeneutic circle “the unity of the understood meanings” should be extended such that with each step the developed understanding is based on the one that emerged from the previous steps. The research presented in this thesis was conducted in a way that meets the principle of the Hermeneutic Circle. The investigation of the various aspects of documentation practices as well as understanding what actually is documentation in scientific software development came from analysing bits of data which were in an iterative way put together to form the big picture. This big picture provided then more context to investigating the separate elements. Inductive coding applied to all qualitative data collected during all four studies reported in this thesis meant that the analysis of the data started from the fragments of information which were then built together to gradually form the themes. The detailed description of inductive coding used in this research is described in section 4.5.1 of this chapter.

The second principle is that of Contextualization. It states that interpretive inquiry should always consider the social and historical context of the phenomenon under investigation. Failure

to do so may lead to complete misunderstanding of the investigated topic. Since the studied phenomenon may be set within different contexts the choice of context which needs to be understood depends on the audience the researchers want to address and the story they want to tell. In the research reported in this thesis one of the key steps was to understand the context in which scientific software development happens, obtain background information about the software documentation practices reported by the participants and collect information about the history of the investigated process. Thanks to the qualitative data obtained in each of the conducted studies the principle of Contextualization was met.

The third principle is that of Interaction Between the Researcher(s) and the Subjects. This principle assumes that the data is produced as a part of the interaction between the researchers and the participants. The presence of the researcher(s) and their conversations or any other activities in which they may engage with the participants contributes to the data. It also means that the participants themselves should be recognised as interpreters and analysts themselves. The information which they provide the researcher is the kind of information which they think is most sought by the researchers and best addresses the researchers' questions. In the research reported in this thesis all reported findings are certainly filtered through the participants' and the researcher's preconceptions and assumptions (as these are simply impossible to be completely avoided). However, the interview questions used for most of the data collection were discussed with the supervision team consisting of researchers with long and extensive experience in conducting empirical studies in software engineering in particular in qualitative studies. When needed the interview questions were refined and improved to minimize constraining or influencing the answers of the participants.

The fourth principle is that of Abstraction and Generalization. This principle states that interpretive research attempts "to relate particulars as may be described under the principle of contextualization to very abstract categories" (Klein and Myers, 1999, p. 75). The purpose of the research should be to understand how the findings about the studied phenomenon might be related to more general and abstract concepts. It does not mean that the context of the study should be ignored but rather it should be considered how the outcomes and the conclusions of the study can extend beyond that context. The findings from the research reported in this

research are discussed in relation to the existing literature, including the literature covering various social constructs, such as communities of practice. The findings are compared and contrasted with the outcomes of other studies conducted in different contexts.

The fifth principle is that of Dialogical Reasoning. This principle states that the researchers should avoid any assumptions and prejudices. The researchers should be entirely clear about the intellectual basis which helped to develop the research design and served as the lenses through which the data was collected. Klein and Myers (1999) explain that in hermeneutics constructing understanding starts from the prejudices. The key thing is to make sure that the researchers are aware of these prejudices and confront them as the research progresses. Indeed, researcher's initial assumptions and prejudices were challenged by the outcomes of the studies reported in this thesis. One of the ways to challenge them was to apply inductive coding to the qualitative data (as described in section 4.5.1 of this chapter). Thanks to inductive coding the data spoke for itself rather than was fit into pre-defined categories. In addition to that, the findings from the first main study (reported in chapter 5) were presented during a seminar to the study participants. The feedback from the audience was used to evaluate the accuracy of the findings. The participants who were at the seminar in general agreed with the presented findings and the conclusions which were drawn.

The sixth principle is that of Multiple Interpretations. This principle states that multiple viewpoints on a given phenomenon should be captured and taken under consideration as this helps to understand and examine the influence of the social context on the studied phenomenon. In the studies reported in this thesis the principle of Multiple Interpretations is particularly well met. The collected data comes mainly from interviews with a number of different participants who work in a variety of social contexts. The facts about the participants' backgrounds are recorded and used to discuss their viewpoints and interpretations of the phenomenon under investigation.

The seventh principle is that of Suspicion. This principle states that the facts reported by the participants of the study should be treated with a degree of suspicion and reasonable disbelief.

There may be various reasons why the participants may filter the information they provide. They may want, or are obliged, to protect their organisation's interests. They may assume that some information is not relevant or of little interest for the researchers and thus they may not refer to certain facts which are actually of great value and interest to the researchers. The participants may also feel that certain facts may contribute to a negative image of their organisation or any phenomenon which they are involved in and hence they will not report these facts. All this may happen as an act of good will on the part of the participants. Nevertheless, researchers should always try to question and verify the reported information. For example, researchers may try to confirm the reported information with data from other independent resources and contrast and compare the accounts of different participants. In the research reported in this thesis the facts provided by the participants were not immediately taken for granted but rather compared with the relevant literature which also helped in taking a critical perspective on the participants' accounts.

4.3 Data collection

The data for the research reported in this thesis was collected using a range of methods. As discussed earlier a qualitative research approach was adopted and hence the data provided rich and descriptive information. At the same time all studies had primarily an exploratory character. In particular, the preliminary study reported in chapter 2 and the first study reported in chapter 5 had a very broad focus and aimed to explore the topic rather than, for example test an upfront established hypothesis. The two subsequent studies (reported in chapter 6 and chapter 7) had a narrower scope but still were of an exploratory nature. The methods used for collecting data for the subsequent studies reflect this gradual narrowing of the focus of the overall research. For the first two studies semi-structured interviews with open questions were used in study 2. Then work-based interviews with a more narrowed scope of inquiry were used. Finally, a case study was conducted. All data collection methods are discussed in detail below. The details about the participants and the exact sampling methods are presented in each of the study chapters. The aim of this section is to explain and justify the selection of the data collection methods.

4.3.1 Study 1: semi-structured interviews

Interviews are a data collection method which allows the researcher to obtain rich and descriptive data (Robson, 2002). In exploratory studies which investigate a broad topic area, with the possible aim of narrowing the scope for further research rather than test an upfront hypothesis, collecting such data is key. The research reported in this thesis has, overall, an exploratory character. It investigates documentation production and use in scientific software development. It also attempts to understand the role of documentation in scientific software development. Hence, interviews were the main method of data collection used in all four studies undertaken for this PhD thesis. It should be noted that interviews have also been used as a method for research in software engineering (Hove and Anda, 2005; Seaman, 2008). Work-based interviews were used in study 2 reported in chapter 6. The details of this method are discussed in the next section 4.3.2. For the remaining three studies semi-structured interviews were used.

A further reason for collecting data via interviews was the fact that the topics investigated were often related to past events and there were no or hardly any other data sources available which could provide relevant information. Interviews allow the researcher to gather “historical data from the memories of the interviewees” (Seaman, 2008) and therefore this method appeared to be the most appropriate for the conducted studies. In addition to that, the interviews provide an opportunity to ask not only about specific facts but also to learn about motivations, beliefs, justification for decisions which were made and so on (Robson, 2002). In other words, interviews enable the collecting of potentially a lot of information which helps to understand the overall context of the given phenomenon. The interactive nature of interviews means that if there is a need to probe the participant for more information, a researcher is able to simply ask the participants to elaborate on a particular topic. In exploratory studies such a capability is a great advantage.

The interviews used in all studies were semi-structured. Semi-structured interviews use open-ended questions which allows for unexpected topics to be raised during the interview and that enriches the data (Seaman, 2008). A set of questions and topics to be covered was prepared in the design phase of each of the studies. In the case of semi-structured interviews such a set of questions helps to guide the interview but should not constrain it in any way (Robson,

2002). A considerable level of freedom and flexibility was kept to allow the interviewees to spontaneously bring up new topics to the discussion and to elaborate on things which they felt were particularly important. As was mentioned, the set of questions for guiding the interview was discussed with the supervision team consisting of experienced researchers. The questions were revised in order to avoid leading the interviewees into a particular mindset.

4.3.2 Study 2: contextual inquiry and work-based interviews

In study 2 reported in chapter 6 data was collected using a specific kind of interview: work-based interview. This technique is derived from contextual inquiry which aims to “reveal the details and motivations implicit in people’s work” (Beyer and Holtzblatt, 1999). The goal of study 2 was to investigate the use of documentation for scientific software. One of the main challenges was to capture what sources of information about the software the participants used. That would allow the researcher to understand what actually should be considered to be *documentation* for scientific software. The problems of using traditional (i.e. not work-based) interviews were that: a) the researcher would not know what sources to ask about and b) the participants would simply forget about some sources or consider them completely irrelevant and not mention them at all. Using contextual inquiry techniques (such as work-based interviews) helps to mitigate these risks. The contextual cues may help the study participant to recall important facts and details. As Raven and Flanders (1996, p. 2) state:

Contextual Inquiry is based on the following three principles:

- 1. Data gathering must take place in the context of the users’ work.*
- 2. The data gatherer and the user [the participant] form a partnership to explore issues together.*
- 3. The inquiry is based on a focus; that is, it is based on a clearly defined set of concerns, rather than on a list of specific questions.*

In the documentation use study the participants were interviewed at their desks where they normally work. They were asked to show how they work with different pieces of scientific software, how they solve problems which may occur during that work and how they proceed if they want to complete various tasks related to the software. The focus of the inquiry was to learn where and how the participants find information necessary to perform all the mentioned activities. There was some degree of partnership between the researcher and the participants, in

the sense that when the participant was demonstrating a particular activity, the researcher asked some additional questions about the process and the actions helped the participant to recall the details.

4.3.3 Study 3: case study

Study 3 reported in chapter 7 constituted a self-contained case study. A case study is not a method of data collection but rather a specific approach to conducting research. Runeson et al. (2012) define a case study in software engineering as:

an empirical inquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context; especially when the boundary between phenomenon and context cannot be clearly specified. (Runeson et al., 2012, p. 12)

The objective of study 3 was to describe, explore and understand the whole process of producing (crowd sourcing) documentation by users in scientific software. Such a documentation project existed within the SciPy¹ community and hence it was possible to conduct a case study research focused on the SciPy and NumPy² Documentation Project.

Runeson et al. (2012) refer to Robson's (2002) classification of case study types which includes "exploratory [case study is about] - finding out what is happening, seeking new insights and generating ideas and hypotheses for new research". This indeed was the focus of study 3 with the exception that no hypotheses were generated. However, the outcomes of the study provided a number of ideas for further research. Runeson et al. (2012) add that Robson's exploratory type of case study is similar to an interpretive case study defined by Klein and Myers (1999) as "attempting to understanding phenomena through the participants' interpretation of their context". This approach was adopted in study 3 in which the explanation of the documentation crowdsourcing process was in a large part based on the information coming from the interviews and from the community mailing list archives. Indeed data collected from mailing lists combined with data from other resources may be very useful in researching a community's practice. In her ethnographic study for scientific practice in the biological discipline of system-

¹SciPy is a suite of libraries of numerical routines written in Python; SciPy is widely used and developed by scientists in various disciplines - <http://www.scipy.org/>

²"NumPy is the fundamental package for scientific computing with Python" - <http://www.numpy.org/>

atics Hine (2007) argues that: “Mailing list observation can be a useful part of ethnography of e-science, but it is valuable also to move the observation beyond list boundaries qualitatively and quantitatively” (Hine, 2007, p. 624). Hine (2007) notes that her research approach “was highly specific” to the topic of her study. But she also adds that in general studying online communication and “Web landscapes (...) offers the potential to develop a rich and rounded approach to understanding the experience of e-science”. The SciPy and NumPy study reported in chapter 7 is not strictly a study of how science (or e-science) is done. But its focus, scientific software documentation crowdsourcing is closely related to scientific research done with that software. And the research done with SciPy and NumPy fits largely the definitions of e-science (Hey and Trefethen, 2002).

4.4 Ethical considerations

Before collecting any data the studies conducted for the purpose of this research were granted approval from the Open University Human Participants and Materials Ethics Committee. All participants were provided with a participant consent form with a clear explanation of the purpose and method of the data collection. The consent form is included in Appendix 1 to this thesis. A written consent was obtained from each study participant. All collected data was anonymised to remove any information which could directly identify people or projects.

In the case of the NumPy and SciPy project used as a case study and reported in chapter 7 the permission to use the names of the projects as well as URLs and so on was granted. This proved to be helpful because it would be apparent for anyone in the area that the discussed project was indeed the NumPy and SciPy project. A part of the collected data, the progress reports and a technical overview, also used these names. These documents are publicly available, were published as conference proceedings and explicitly mention the names “NumPy” and “SciPy”. Additionally, the description of the case was easier to write (and easier to follow for a reader) with the ability to use the names “NumPy” and “SciPy” rather than introduce some phony names such as “project X”.

4.5 Data analysis

This section provides a detailed description of the data analysis process. Particular attention is paid to inductive coding and thematic analysis applied to the qualitative data collected in all four empirical studies. Both inductive coding and thematic analysis involve a high degree of the researcher's interpretation of the data. And this interpretation process is relatively implicit (in comparison, to for example, building statistical models). Therefore a very detailed, step-by-step description of the analysis ensures that this process is transparent and clear to the reader. It also aims to demonstrate the rigor which was retained throughout the whole process.

The case study reported in chapter 7 involved collecting and analysis some quantitative data. The analysis process, described at the end of this section, was relatively straightforward. Simple descriptive statistics was generated from the collected data. The discussion of this part of the data analysis is short as it did not involve any complicated statistical tests or models.

4.5.1 Inductive coding and thematic analysis

A thematic analysis (Boyatzis, 1998) was applied to the transcribed interviews as well as, partially, to the data from the mailing lists archive collected in the last study reported in chapter 7. A unit of analysis was a single participant and, in the case of the mailing lists archive, a single email message sent to the list. A unit of coding was a chunk of text from the transcribed interview which could be of the size of: an expression (less than a sentence), one or more sentences or even a whole paragraph (in order to retain the context of what was said). The whole process of coding followed what Boyatzis (1998) describes as a "hybrid" approach combining inductive coding and coding led by findings from the previous research (reported in the literature). Typically the thematic analysis process consists of three stages:

Stage I, deciding on sampling and design issues;

Stage II, developing themes and a code;

Stage III, validating and using the code. (Boyatzis, 1998, p. 29)

For this study Steps I, II and III were implemented in the following way:

1) In Stage I for developing codes inductively it is recommended that subsamples of the analysed material are selected and coded first (Boyatzis, 1998). The subsamples should be selected based on criterion-referencing. That is, if the research is focused around a particular issue ("a dependent variable"), the subsamples should represent different aspects of the issue, to enable

a researcher to later compare the codes developed from contrasting subsamples. However, in exploratory research in which “the dependent variable” is not defined (i.e. the patterns and the relationships between things will only be discovered), selecting subsamples for developing codes may not be possible. That was indeed the case in this study which explored the broad topic of documentation production in scientific software development. Hence in Step I no subsamples were selected and the codes were developed using the whole data set (all interview transcripts).

2) Stage II involved working very closely with the data and building up codes and then themes. Primarily an inductive coding was applied. That is, no “code book” was developed prior to coding and the codes emerged directly from the data. As Boyatzis (1998, p. 31) defines it:

A good thematic coding is one that captures the qualitative richness of the phenomenon. [...] A good thematic code should have five elements:

- 1. A label [...]*
- 2. A definition of what the theme concerns [...]*
- 3. A description of how to know when the theme occurs*
- 4. A description of any qualifications or exclusions to the identification of the theme*
- 5. Examples, both positive and negative, to eliminate possible confusion when looking for the theme*

The codes were developed in the following way:

Step 1: In the course of reading the whole transcript chunks of text (units of coding) were marked and a code was assigned to them. This means that at the beginning of the coding procedure there was almost exclusively code generation (i.e. usually only a new code was assigned to a unit of coding). As the coding progressed, the codes were sequentially reused.

The following table shows example codes generated from fragments of the transcribed interviews.

Study	Coded fragment	Code	Definition
Preliminary	“... my colleague wanted to produce something which could be sold proactively and not just wait for somebody to make an inquiry , so to speak. That’s how it started... ”	Software-Commercialisation -Start-Reasons	The reasons for starting the software commercialization process.

Study 1	“...I do sometimes step back and write down the steps that need to be taken... You start from a principle that you are going to write a module. If you know the code you can actually do that quite easily. Actually writing down - I do write notes to myself but I never write formal specifications that someone else would read or I would expect someone to read and understand...”	AT-Documentation-Requirements	Documentation related to requirements and software specification after transition from the scientific end-user development into developing for a wider community.
Study 2	“...when I first started out it was more a case of using this [the namespace] and the README file and FAQ and all the other stuff. But then I needed to start making my own modifications , so I was quite quickly into messing around with the source code . I guess I’ve just got used to looking at it right now because I’ve been messing with this...”	User-to-Developer	The circumstances in which the interviewee changed from being an end-user into being a user-developer.
Study 3	“...in the spirit of Election Day (and to try out our new polling plug-in), I set up a poll for documentation formats on [the website]. You should be able to see it on the front page, or you can go to [direct link to the poll]...”	Documentation-Formats-Tools-Poll	Selecting the documentation formats and tools using polls / voting.

Table 4.2: Examples of codes generated during the coding process. The emboldened words are the words (or expressions) which allowed the researcher to develop the code.

Step 2: The code refinement. All codes were double-checked to ensure that the units of analysis marked with the particular code fitted the label and the code definition.

Study	Coded fragment	Code version 1	Code version 2
Preliminary	“...I’d say he has some scientific background. He has a PhD in physics. He worked on other types of projects like semi-conductors. But he is not a material scientist. So he doesn’t have much say in the actual algorithmic equations processes we use to get answers out of the system...”	Research-Degree-Physics: The research degrees which the team members had.	Software-Development-People-Background: The background (education, experience) of people who have been engaged in the software development.
Study 1	“...we only started fetching user feedback rather later. What we did do is that we have a couple of starting points, there was Sparky which was a very popular program. We looked at that and thought people liked this and that was good and let’s see how they do it. And do something similar...”	Benchmarking-Users-Expectations: The ways of finding out about users’ expectations based on other software examples.	AT-Gathering-Requirements: The ways of gathering users’ requirements and needs, after transition from the scientific end-user development to developing for a wider community.
Study 2	“...I don’t think you can [assess the correctness of the software]. I think that’s the bottom line. You need something to compare against. Otherwise you’re stuck with intuition. It does do something which looks sensible. You could compare the generic behaviour, the qualitative behaviour with some other theory or related theory or related model. But that would again need to be qualitative. It depends what claim you are making with your code. So with these two codes the claim is that these are exactly the numerical methods that will give you the exact answer...within the statistical error of using the sub-diagrams that model multi-core...”	Development-Software-Assessment: Ways of assessing whether the software does what it is expected to/supposed to do.	Software-Development-Testing: The overall approach to testing the software at any stage of software development.
Study 3	“...but what about equations? Seems to me that documenting numerical and scientific libraries needs a good way of expressing equations and formulae. Does ReST support this? (My previous looks at it didn’t show much in that way)...”	Documenting-Mathematics: The ways of capturing the mathematical details in the documentation.	Documenting-Formulae: The ways of capturing and expressing formulae in the documentation.

Table 4.3: Examples of refined codes

Step 3: Merging the codes. During the process of refinement there were some units of coding which were coded with more than one code. This helped to identify the codes which actually marked the same concept/problem and were redundant. Such codes were merged and the labels as well as the code description were appropriately amended.

Study	Code 1	Code 2	Merged code
Preliminary	Software-Development-Windows-OS	Software-Development-Linux-OS	Software-Development-OS

Table 4.4: Example of merging codes in the data analysis for the preliminary study.

In the above example, these two codes were merged as this level of granularity (that is, whether the software was developed for the Windows or Linux) was not needed for this study.

Study	Code 1	Code 2	Merged code
Study 1	AT-Challenges-Keeping-Userbase	AT-Developers-Users-Relationship	AT-Developers-Users-Relationship

Table 4.5: Example of merging codes in the data analysis for the study 1.

In this example, code 1 was merged into code 2. The challenges of keeping the userbase was mentioned only by 2 participants and it was discussed within a wider topic of user-developer relationship. “AT” means “After Transition from the scientific end-user development to developing for a wider community”.

Study	Code 1	Code 2	Merged code
Study 2	Sources-Frequency-of-Usage	Sources-Sequence-of-Usage	Sources-Use-Patterns

Table 4.6: Example of merging codes in the data analysis for the study 2.

In the example above code 1 and 2 were merged and a new code was created. Combining together the information about the frequency of use of the documentation and the sequence in which it was used allowed for understanding the whole pattern of using the documentation that was more informative than just considering separately the frequency and the sequence.

Study	Code 1	Code 2	Merged code
Study 3	LaTeX-ReST-Word-for-Tables	LyX-advantage	Documentation-Tools

Table 4.7: Example of merging codes in the data analysis for the study 3.

In the example above code 1 and 2 were merged and a new code was created. It was not necessary to code at this level of detail what were the exact advantages of different tools suggested by the community members to be used for documentation.

3) In Stage III the codes were validated by collaboration within the supervisory team. A sample of the units of coding (quotations) from the data and a list of codes was prepared. The codes

were then assigned to the units of coding by the supervisory team who did not do the earlier coding and did not develop the codes. This selection of codes was then compared with the codes that were originally assigned to the given units of coding. If there were any discrepancies between the originally assigned codes and the codes assigned during the validation procedure, all these discrepancies were discussed. Typically, providing more context to the units of coding was sufficient to defend the choice of the originally assigned codes.

Once the codes were ready they were sorted and grouped into clusters. Building themes was informed by the clusters of the data but also by the research sub-questions for the studies. This hybrid approach of using an inductive way to form the themes which use the code clusters along with some preexisting “thoughts, ideas and perceptions” is recommended by Boyatzis (1998). These pre-existing “thoughts, ideas and perceptions” emerged as a result of literature review and some initial informal analysis which occurred when conducting and then transcribing the interviews. The themes formed were then used to investigate the patterns emerging from the data and to construct the findings reported later in this thesis.

Study	Codes	Clusters
Preliminary	Career-Change-Towards-Software-Development; Self-perception; Software-Commercialization-Change-Team; Software-Commercialization-Finding-Partners; Software-Development-Collaboration; Software-Development-People-Background; Software-Start-People	Software Developers
Study 1	Background-Software-Education; Experience-Academic-Software-Development; Experience-Non-Academic-Software-Development	Background Software Development
Study 2	Sources-Reliability-Criteria; Sources-Reliability-Hierarchy	Sources Reliability
Study 3	Arguments-for-Crowd-Sourcing; Choosing-Documentation-Tools-by Users; Suggestion-New-Users-Engagement; User-Suggestion-to-Contribute-to-Doc	Ways-to-Engaging-Community

Table 4.8: Example of grouping codes into clusters.

4.5.2 Quantitative analysis

The core of the data for this thesis consists of rich qualitative data. This data came from semi-structured interviews, work-based interviews and artifact walkthroughs as well as the archive

of three mailing lists and the progress reports (Harrington, 2008; Harrington and Goldsmith, 2009; Van der Walt, 2008). In addition, the third main study reported in chapter 7 included some quantitative data analysis. Firstly, quantitative analysis was applied to the data from the mailing lists archive. The aim of this analysis was to compare the outcomes of the quantitative analysis with the findings from the qualitative data analysis. Quantification of qualitative data is possible and typically preceded by qualitative analysis of this data in order to understand what main categories can be extracted from the data (Seaman, 2008). The posts on the mailing lists were classified according to the categories which emerged from analysing both the mailing lists contents and the interviews conducted within the case study. This categorisation allowed the researcher to then develop simple descriptive statistics. The findings of this analysis were then helpful in establishing whether certain events and topics which were reported in the interviews as particularly important and intensively discussed at the time were actually present in the discussion on the mailing lists. Another part of the data which was quantitatively analysed was the data from the documentation project server. This data was purely quantitative, that is, contained information about the number of people who signed up for the documentation project, the number of lines of documentation produced and so on. Thus only quantitative analysis was possible on this data.

Adding the quantitative analysis to the large body of qualitative data aimed to make the research more robust and support the findings coming from qualitative analysis with the findings that were obtained from another source. The goal was not to produce statistically significant results but rather to show that other sources of data analysed in a different way corroborates the outcomes of the study. Also, any discrepancies between these findings provided an opportunity to reexamine the data and to discuss why these differences occur.

4.6 Limitations

Examining the limitations of empirical studies enabled the researcher to establish the validity of the conducted research. Empirical studies need to be valid if they are to be considered as making contributions to existing knowledge (Easterbrook et al., 2008). As Robson (2002, p. 66) puts it: “Validity is concerned with whether the findings are ‘really’ about what they appear to be about”. This section discusses the limitations of the studies which is reported in this thesis

in the light of the relevant literature in the areas of research design and methodological issues. The ways in which the limitations of the studies were addressed are also presented. In the first subsection, the general limitations of qualitative research are considered. The second section focuses on assessing the study validity.

4.6.1 General limitations to a qualitative approach

It is not the purpose of qualitative approach to provide statistically significant findings but rather to provide an in-depth insight into the phenomena under study (Robson, 2002). Since qualitative studies typically include a relatively small number of subjects, it inevitably raises questions about generalizability. The concern is whether the conclusions can be extended outside the particular study settings. Selection of the subjects of study is one of the key factors which can affect generalizability. The limitation is that the results may not be generalizable beyond the studied case.

In order to address this limitation with generalizability beyond the studied case, the study participants as well as the case study were chosen so that they could provide information relevant to understanding the role of documentation in the context of scientific software development. And even though it cannot be claimed that the subjects of the studies presented in this thesis constitute a statistically representative sample, they are certainly not an atypical group within the whole community of scientists developing and using software.

Objectivity is another concern when it comes to the potential limitations for qualitative research (Robson, 2002). Interpretive inquiry accepts that there is no one and objective truth about the researched phenomenon (Klein and Myers, 1999). The complexity of the real world settings, the multiple factors which contribute to how the context is shaped and perceived and the variety of different lenses which can be chosen to apply to the study means that drawing an objective picture is not possible. However, thanks to retaining the rigour in the data analysis process, the interpretation bias in this study was minimised. Also an explicit description of the data analysis steps allows the reader to assess in-depth the quality of the research and audit all steps undertaken during data collection and analysis.

4.6.2 Assessing validity

One of the popular ways to “classify aspects of validity and threats to validity” (Runeson et al., 2012, p. 71) is a classification scheme used by Yin (2009). However, this approach is more relevant to a theory driven, positivist research (Easterbrook et al., 2008). As Easterbrook et al. (2008) say “in the constructivist stance, assessing validity is more complex”. Following Creswell’s (2003) Easterbrook et al. (2008, pp. 306-307) list eight strategies for for improving the validity:

- “1. *Triangulation (...)*
2. *Member checking. (...)*
3. *Rich, thick descriptions (...)*
4. *Clarify bias (...)*
5. *Report discrepant information (...)*
6. *Prolonged contact with participants (...)*
7. *Peer debriefing (...)*
8. *External auditor”*

5 out of 8 of these strategies were applied for this research.

1. Triangulation - for the case study reported in chapter 7 data was collected from different sources and then compared to build a coherent picture. The triangulation between the studies was not possible as all each study had a different focus and different kinds of participants.
2. Member checking - the findings from study 2 were presented at a seminar at which more than a half of the participants of the study were present. They agreed with the findings presented occasionally adding some comments.
3. Rich, thick descriptions - all chapters presenting the conducted empirical studies include detailed information about the data collection process and a significant amount of data itself (the quotes) is presented.

5. Report discrepant information - any findings which contradict the findings previously reported are not only reported but emphasized in the study discussion.

7. Peer debriefing - this research has been supervised by a team of three experienced researchers who conducted a number of empirical studies in software engineering (including qualitative interpretivist studies). All steps of the research were discussed with the supervision team. The comments and feedback was used to refine and improve the work.

Strategy 4. "Clarify bias" was not applied because the bias was not identified and hence, it was not possible to discuss and clarify it. Strategies 6. "Prolonged contact with participants" and 8. "External auditor" were not applied due to pragmatic reasons, that is lack of relevant resources to support introducing such strategies.

4.7 Summary

A qualitative approach was chosen for conducting this research because the study had an exploratory character and that demanded rich data and approaching the topic without pre-defined hypothesis and assumptions. The themes, concepts and ideas, emerged from the data. Qualitative data collection and analysis allowed the researcher to form these themes and guide further work. Rich and descriptive data was also most appropriate for addressing the broad research questions. The seven principles of interpretive inquiry were clearly met by the study positioning this research within a recognized epistemology. The dominant method for data collection was a semi-structured interview. Third study involved: the mailing lists archive, the progress reports and the logs from the project server. All qualitative data was analysed using thematic analysis. Quantitative data was used to create some descriptive statistics in order to compare it with the results of qualitative analysis. The limitations of the study involved mainly concerns about generalizability and objectivity. The former was addressed by finding participants and a case study which were typical for the context of the studied phenomenon. The latter was ensured by a rigorous and well documented data analysis process. At the same time it is recognised that in interpretive studies the concept of "one and objective" truth does not exist.

Chapter 5 Study 1: scientific software documentation production

5.1 Introduction

The preliminary study (reported in chapter 2) found that lack of or poor software documentation affected further software development and prompted questions about documentation production in scientific software development. How is software documentation produced by scientist-developers? What should actually be considered as documentation in scientific software development? What information about software do scientist-developers capture? How do documentation practices change when the context changes from end-user development to developing for re-use by other scientists? What is the role of the user community in documentation production? These questions form the main focus of the research reported in this thesis, and they led to Study 1, presented in this chapter, which explored how documentation is produced in the scientific software development context.

The study focused on the real-life documentation practices of scientists who used to develop software in the scientific end-user context and then moved on to develop software for a wider community. These scientists were selected specifically in order to collect data which might capture differences in documentation practices between the two contexts. 22 interviews were conducted with 21 scientist-developers and one professional software developer developing different kinds of scientific software. The three main research questions which shaped this study were:

- *What are scientist-developers' documentation practices in the scientific end-user development context?*
- *What are scientist-developers' documentation practices in the context of developing for a wider community of scientists?*
- *What is the role of the user community in documentation production?*

Section 5.2 discusses the methodology adopted for this study. Section 5.3 presents the findings, divided into three main sections, corresponding to the three research questions. The subsequent section discusses the findings in terms of the relevant literature. The chapter ends with a summary.

5.2 Methodology

This section presents the research approach, first providing details about the participants of the study, and then describing the processes of data collection and data analysis.

5.2.1 Participants

The participants for the study were selected using convenience sampling (Marshall, 1996). In order to inform the research questions, five selection criteria were established:

1. They were scientific software developers, in a computational science domain (i.e., a computationally-intensive domain involving development focused on algorithmic complexity);
2. they had a degree in a computational science domain;
3. they previously developed scientific software for themselves, and then they moved on to develop software for a potentially wide group of scientists (possibly including themselves);
4. they did not have a formal degree/education in software engineering and/or computer science;
5. they did not have significant experience in working as a professional software developer.

9 participants were identified via the participants of previous studies conducted by one of the members of the supervision team. A call for participation was advertised on two websites: the Software Sustainability Institute¹ and Software Carpentry²:

- 19 participants were scientists who met all of the criteria;
- 2 participants were scientists with an additional degree/background in computer science, but who currently worked as scientist-developers producing scientific software for a potentially wide group of scientists;
- 1 participant was a professional software developer engaged in a scientific software development project.

¹www.software.ac.uk

²www.software-carpentry.org

Even though three participants did not meet all criteria, their interviews were still used to inform the research. During the analysis it became clear that their experiences and opinions brought additional insights and were helpful in identifying and understanding the main themes emerging from the data. Therefore, the data from these three participants were included in the main data collection.

ID	Domain	Work position	Training in software development	Experience in software development	Was end-user developer	Current work in software development
A	Particle physics	Academic/ Research (University Professor)	Self-taught using books and internet resources.	Before going to university used to work as ASSEMBLY programmer. End-user developer who now develops software used for others but still develops for himself.	Yes	Currently 30% of his work is software development. Several projects: typically in small development teams. Software given away to be used by others.
B	Biochemistry	Academic/ Research (University)	A 3-weeks course in FORTRAN as undergraduate. Then mainly self-taught from books and online resources.	End-user developer at the beginning of his academic career. Now almost exclusively develops for a wider community.	Yes	For over 10 years most of the work in one main project, in collaboration with 2 other core developers and a few external ones. Software free for academics and sold to industry.
C	Biochemistry	Academic/ Research (University)	Self-taught from books and internet resources. Currently works on a book about Python and teaches programming to students.	End-user developer at the beginning of his academic career (mainly during his PhD). Now almost exclusively develops for a wider community.	Yes	For over 10 years most of the work in one main project, in collaboration with 2 other core developers and a few external ones. Software free for academics and sold to industry.
D	Quantum Chemistry	Researcher (Research Institution)	A semester course in C++ during Master's degree.	End-user developer during his Master's degree. Then moved on directly to developing software for others.	Yes	Involved in several projects sometimes works on more than one project at the same time.
E	Molecular Physics	Researcher (Professor; Research Institution)	Primarily self-taught using books and internet resources.	End-user developer at the beginning of his career but still occasionally developing for own use. Involved in many projects aimed at a wider community of users also as a project leader.	Yes	Currently leading a large project which is an inter-institutional collaboration aiming to refine and improve a big scattered legacy code and turn it into a software suite.
F	Physics	Researcher (Research Institution)	Undergraduate studies in Computer Science. A diploma in programming technology.	8 years working as a professional software developer. Due to his first degree and extensive work experience not v. good example of transition. But works with scientists undergoing transition and is occasionally an end-user developer himself.	No	Sometimes involved in many projects at the same time: specializes in High Performance Computing.
G	Physics / Computational Fluid Dynamics	Researcher (Research Institution)	A FORTRAN course during undergraduate degree.	End-user developer at the beginning of his academic career. Then for a short time worked in commercial software development before moving back to academia to develop software used by others.	Yes	Typically involved in one main project but may contribute to other code development at the same time.

ID	Domain	Work position	Training in software development	Experience in software development	Was end-user developer	Current work in software development
H	Physics / Computational Biology	Researcher (Research Institution)	Primarily self-taught using books and internet resources.	End-user developer during his PhD studies. However, during that time already worked on source code used by others. Still occasionally develops for his own use.	Yes	Currently redeveloping codes already existing (created by other scientists). The main current project may have around 700 international users.
I	Physics	Researcher/ Academic (University)	Originally studied computer science (spec: cryptography and information security) then switched to physics.	Initially did not develop that much for himself and very quickly moved on to projects aimed at many users. Worked as a programmer for a large physics project.	Yes	Involved in several projects with a focus on one large international project which aims to deliver a extensive software suite for running experiments.
J	Signal processing/Audio research	Researcher/ Software Developer (University)	A Master's degree in computer science.	Extensive experience in commercial software development as well as in working on software development for or in collaboration with scientists.	No	Currently involved in one main long-term project which aims to support (in different ways) scientists developing software.
K	Signal processing	Researcher/ Software Developer (University)	A course in programming during postgraduate studies. Apart from that mainly self-taught using books and internet resources.	End-user developer during his PhD. Developed software in a spin-off from his original university. Then for a short time worked in a commercial software development company.	Yes	Currently involved in one main long-term project (as a software developer and researcher) which aims to support (in different ways) scientists developing software.
L	Physics	Researcher/ Academic (University)	A course in FORTRAN during undergraduate studies.	End-user developer during her PhD. Then moved on to working with legacy code which was sometimes used by others.	Yes	Currently working in a large project which is an inter-institutional collaboration aiming to refine and improve a big scattered legacy code and turn it into a software suite.
M	Computational fluid dynamics / Physics	Researcher (Research Institution)	Learnt basics of computer programming in high school. Then primarily self-taught.	End-user developer during his postgraduate studies then started a job at a university but as a software developer in a research project which was collaboration with commercial companies.	Yes	Involved in a few projects with the main one being a large software suite for physics. Specializes in parallel code.
N	Physics/HPC	Researcher (Research Institution)	Learnt some programming in high school. Then mainly self-taught.	End-user developer during his undergraduate and postgraduate studies. Then joined computational science projects which aimed to develop for a wider community. Experience in both development and managing projects.	Yes	Involved in a few projects; but also still occasionally develops for his own use.

ID	Domain	Work position	Training in software development	Experience in software development	Was end-user developer	Current work in software development
O	Crystallography / Computational Chemistry	Researcher (Research Institution)	A programming course during undergraduate studies. The primarily self-taught.	End-user developer during his PhD (the software was also used by a small group of colleagues). Then joined computational science projects which aimed to develop software for a wider community of users.	Yes	Involved in a number of projects, primarily in computational chemistry and computational materials.
P	Computational Chemistry / HPC	Researcher (Research Institution)	Primarily self-taught. Had some formal training in MPI.	End-user developer during his PhD adapting already existing code written by another researcher. Then joined computational science projects which aimed to develop software for a wider community.	Yes	Currently working primarily on software for particle physics simulation. Partially involved in other projects too.
R	Physics/HPC	Software Developer/Researcher (Research Institution)	Learnt some programming at school and at the university. Then mainly self-taught.	End-user developer during undergraduate and postgraduate studies. During his studies and for a few years after graduation worked as a developer for a commercial company doing modelling. Then joined computational science projects aiming to develop for a wider community of users.	Yes	Involved in a number of projects mainly HPC; some projects are international.
S	HPC	Researcher/ Manager (Research Institution)	Primarily self-taught. Some formal training in software project management.	End-user developer during his PhD. Then joined computational science projects aimed to develop software for a wider community of users- initially was a developer then moved on to a managerial post.	Yes	Involved in a number of projects mainly HPC; some projects are international.
T	Geology	Researcher/ Academic (University)	A course in programming in FORTRAN during undergraduate studies. Then mainly self-taught.	End-user developer during her postgraduate studies. Recently started contributing to the software which is used by others.	Yes	Currently working with existing software contributing various modules. The software has a number of international users.
U	Pressure measurements	Freelancer	A course in programming during undergraduate studies. Primarily self-taught using books and internet resources.	End-user developer during his postgraduate studies. Then worked for a commercial company and gain some experience in gathering requirements.	Yes	Currently works on his own project as a freelancer. The software will be freely available for the research community.

ID	Domain	Work position	Training in software development	Experience in software development	Was end-user developer	Current work in software development
V	Biochemistry / Crystallography	Researcher/ Academic (University)	Primarily self-taught. Started learning programming before beginning his studies.	End-user developer during his postgraduate studies. During his PhD used and redeveloped different bits of software produced by other scientists. Then joined a team developing scientific software for a wide group of users. The team consisted mainly of scientists but there were also professional developers involved.	Yes	Currently supports/maintains a large scientific software suite.

Table 5.1: Study1 participants

5.2.2 Data collection

The data were collected using semi-structured interviews. This method allows for collecting rich and descriptive data and is often used in exploratory studies (Robson, 2002). Additionally, this method enables a researcher to elicit information not only about the current situation but also about past activities (Seaman, 1999). A detailed discussion of the data collection methods and the overall research design for this and other studies reported in this thesis is presented in chapter 4.

A list of questions and topics which needed to be covered was prepared before the interviews. However, much of the initiative for guiding the interview was left to the interviewees. The purpose of having the set of topics and questions was to ensure that everything that needed to be covered during the interview would be discussed. The interviewees were not shown the list of topics and questions, so that they could bring up topics which might not have been considered before and hence were not included in the list. This approach also ensured that the exploratory character of the study was retained.

The shortest interview lasted about half an hour and the longest was over one hour and a half. Most interviews lasted around 40-50 minutes. Two interviews were conducted over the phone, and the rest were face-to-face. Face-to-face interviews were conducted at the participants' organisations but not always at their exact workplace. All interviews were audio-recorded and then transcribed.

5.2.3 Data analysis

Thematic analysis (Boyatzis, 1998) was applied to the collected data. The interviews provided rich and descriptive material, which provided information about the context in which the reports were embedded and supported examination of how the components of the complex relationships, events and activities fit together (Robson, 2002). The analysis was inductive: the findings emerged from the data. That is, instead of applying a pre-defined set of topics to categorize the data, topics were generated using inductive coding (Fereday and Muir-Cochrane, 2008). The codes were then grouped in order to form themes discussed further in this chapter:

Theme	Codes
Documentation in the scientific end-user development context	<p>Before-Transition(BT)-Documentation-Technical-Specification</p> <p>Before-Transition(BT)-User-Manuals</p> <p>Before-Transition(BT)-Planning-Development</p> <p>Differences-Documantation-Before-Transition(BT)-vs-After-Transition(AT)</p> <p>Before-Transition(BT)-Developer-User-Relationship</p> <p>General-Comments-Documantation-Scientific-Software</p> <p>Problems-Documantation</p>
Documentation after the transition from the scientific end-user development to developing for a wider community	<p>After-Transition-(AT)-Challenges-Development-Correctness</p> <p>After-Transition(AT)-Documantation-Planned-Development</p> <p>After-Transition(AT)-Documantation-Presentation-Papers</p> <p>After-Transition(AT)-Documantation-Requirements</p> <p>After-Transition(AT)-Documantation-Technical-Software</p> <p>After-Transition(AT)-Problems-Technical-Documantation</p>
Documentation for software users	<p>After-Transition(AT)-Documantation-General-for-Users</p> <p>After-Transition(AT)-User-Manuals</p> <p>After-Transition(AT)-Problems-Documantation-for-Users</p> <p>Differences-Documantation-Before-Transition(BT)-vs-After-Transition(AT)</p> <p>After-Transition(AT)-Developers-Users-Relationship</p>
Documentation for oneself	<p>Before-Transition(BT)-Documantation-for-Oneself</p> <p>After-Transition(AT)-Documantation-for-Oneself</p> <p>After-Transition(AT)-Problems-Documantation-for-Oneself</p> <p>Differences-Documantation-Before-Transition(BT)-vs-After-Transition(AT)</p>

Documentation shared with other developers in the team	<p>After-Transition(AT)-Subcontracting-Documentation</p> <p>Collaboration-Scientists-Software-Engineers</p> <p>After-Transition(AT)-Documentation-Team-Communication</p> <p>After-Transition(AT)-Problems-Documentation-Team-Communication</p> <p>After-Transition(AT)-Work-Organisation-Communication-Collaboration</p>
Learning how to document	<p>After-Transition(AT)-Documentation-Tools-Learning</p> <p>After-Transition(AT)-Documentation-Tools-Usage</p> <p>Collaboration-Scientists-Software-Engineers</p> <p>Learning-About-Documentation</p> <p>Background-Software-Education</p>
Perceived benefits from producing documentation	<p>Inspiration-Documentation-Changes</p> <p>Inspiration-for-Development-Management-Changes</p>
Documentation production and the user community	<p>After-Transition(AT)-Gathering-Requirements</p> <p>After-Transition(AT)-Developers-Users-Relationship</p> <p>Before-Transition(BT)-Developer-Users-Relationship</p> <p>After-Transition(AT)-Challenges-Keeping-Userbase</p>
Perceptions of the user community	<p>Before-Transition(BT)-User-Types</p> <p>User-Types-After-Transition(AT)-End-User-Developers</p> <p>User-Types-After-Transition(AT)-End-User</p>
Feedback from the user community	<p>Inspiration-Documentation-Changes</p> <p>After-Transition(AT)-Documentation-Desirable-Improvements</p> <p>Inspiration-for-Development-Management-Changes</p> <p>After-Transition(AT)-Challenges-Keeping-Userbase</p> <p>Problems-Documentation</p>

Documentation and assumptions about users' knowledge	User-Knowledge-Assumptions After-Transition(AT)-Gathering-Requirements User-Types-After-Transition(AT)-End-User-Developers User-Types-After-Transition(AT)-End-User
--	--

Table 5.2: Codes forming themes in Study 1- Documentation production

As this study was of an exploratory nature, the aim was to find new information which had not been reported in the literature. Therefore inductive coding was the most appropriate approach. Moreover, because few publications covered scientific software documentation (as discussed in Chapter 3), it would have been difficult to build a comprehensive list of pre-defined topics to apply to the data. The analysis was conducted using the NVivo¹ software package.

5.3 Findings

This section presents the finding. Section 5.3.1 describes the production of documentation in the scientific end-user context. Section 5.3.2 discusses the production of documentation after the transition to developing for re-use by a community of scientists. Several sub-topics are considered: production of documentation for different users, sharing documentation within the development team, learning how to document and perceived benefits of documentation production. Section 5.3.3 describes the relationship between documentation production and the user community. The findings focus on perceptions of the user community, feedback from the user community, and assumptions about the users and the level of their knowledge.

Each of the sections includes examples of data as evidence to support the reported findings. Since all data were anonymised, the quotations do not contain any names or other details which may identify the participants. In some cases, more information about the findings is given in the description.

5.3.1 Documentation in the scientific end-user development context

All but 3 of the 22 scientists interviewed started their experience in programming as end-user developers. Typically the interviewees reported that they developed software for their own use during their postgraduate studies. 3 scientists stated explicitly that they still occasionally develop software for their own purposes, although most of the applications which they write are aimed at a scientific community and not only for themselves. Usually the software which they wrote for themselves was of a significantly smaller scale and of a different applicability than the software they developed for use by others.

¹NVivo - Software package for qualitative data analysis; www.qsrinternational.com

If software was produced as part of a postgraduate degree, it was hardly documented at all. What was captured typically was the underpinning scientific model which was discussed in the thesis.

I did some commenting on my PhD code [but] I never felt the need to actually go ahead and document, it because I knew that once I finish, I basically wouldn't need to use it. [Interviewee P]

Yet, after gaining more experience in developing software, writing documentation was perceived as a process which could help a developer revisit and rethink the implementation ideas.

I never felt the need to document it [when I developed code for my own use]. In hindsight I think it would have been a good idea because it makes you think about what the code is actually doing and if it's a good idea to have the code do certain things. It would have been better if I added documentation but at that time I never needed it. [Interviewee V]

One interviewee estimated that, if he had documented his work better, he would have required only a third of the time it had eventually taken him to implement certain changes (compare: Quote 5-1).. An explicitly-mentioned reason why documentation was not provided during the scientific end-user development was the lack of time for writing it. The main goal for writing the software was to obtain the scientific results.

Once you get the code working and it does whatever you want it to do, you don't want to waste your time documenting. The very typical thing that you do is you say "Right I have the code working, now I am going to do the calculations that I want to do. I will come back to write the documentation and comment it later on". The "later on" never happens. I can understand why people leave it and cannot be bothered. [Interviewee L]

The only documentation which appeared to be present consistently in the scientific end-user development context was comments in the source code. One of the participants reported that, when he developed software for his own use, he commented differently, and more economically.

If I am developing for myself, documentation would be a lot lighter. I can just put in a comment line "watch the minus sign here" because when I read it I understand what I meant. [Interviewee A]

Another interviewee whose native tongue was not English used to comment her source code in her mother tongue when she was developing software for her own use. This habit was so strong that, when she moved on to developing software for use by others, she initially still kept writing in her first language (compare: Quote 5-2).

The reliability of comments in the source code, both produced by oneself or by someone else, was not assumed.

The documentation fundamentally is always the code. It doesn't really matter what I or somebody else wrote in a comment. Either on paper or next to a particular line of code. The chances are it wasn't implemented right. I'm gonna have a look at it to see what it does. [Interviewee U]

To summarise, documentation production in the scientific end-user development context was limited. A popular way of documenting was writing comments in the source code. However, these comments were meant to be understandable for the scientific end-user developers themselves. Documentation was limited for several reasons: because there was no expectation that the software would be reused in any way, because of a lack of time, because developers expected to read the code even if there were comments. Whilst technical documentation was scarce or non-existent, information about the scientific model underpinning the software developed in the scientific end-user context was captured.

5.3.2 Documentation after the transition from the scientific end-user development to developing for a wider community

The transition from scientific end-user development to developing for others brought a realisation to the scientist-developers that they would have to maintain the software which they produced. Without documentation, maintenance was extremely difficult, if not impossible.

It doesn't take too many times of doing it before you realize that the next person to come and maintain the code...it is almost certainly going to be you, in 6 months time, when you have forgotten all things that you figured out today. And you are going to have to figure them out again because you were too much in a hurry to make a decent set of notes. [Interviewee U]

The transition to developing for others involved a number of changes in documentation practices. Providing software for a scientific community meant that the scientist-developers had to produce documentation for users. The participants reported that they captured some information about the software strictly for their own use. However, there were other documents which were shared within the development team. The knowledge about documenting software was related to, for example, the style of documentation or the contents which it should include. The following sections provide more detailed accounts of these aspects of documentation practices.

5.3.2.1 Documentation for software users

19 of the 22 participants talked about some kind of user documentation that they produced when they developed software for a community of scientists. When the software was written in the scientific end-user development context, documentation for users was basically non-existent. There were no comprehensive user manuals, because the expectation was that only the scientist-developer used the software. If software was used by people other than its developer and, for example, by a few lab colleagues, a brief README file was sometimes prepared. Such a file typically contained very basic information on how

to set up the software. The transition from scientific end-user development to developing for others meant that there were users who would need documentation in order to be able to use the software.

One of the participants reported that the main part of the user manual which he produced included information telling the reader how to run the program. Since some users were interested in engaging with the source code, technical documentation was also provided (compare: Quote 5-3).

Documentation for end-users was sometimes enriched with short exercises that would help the users get up to speed with the software. These exercises were a kind of step-by-step instruction to show the users how, for example, to conduct a particular simulation. Two participants who worked for over ten years on a very mature software package provided a set of tutorials which were available online. Most of these tutorials were originally prepared for hands-on workshops which they ran on regular basis.

One of the above-mentioned interviewees spent around six months completely rewriting the user documentation. This activity was inspired by the outcome of a survey which was conducted among the software users. The users wanted more and better documentation. In general, the interviewee did not perceive writing documentation for users to be any kind of a major burden. The main challenge, according to the interviewee, was finding time to complete the task.

I knew what to write. I quite like writing. I just need to clear my mind sit down and do it. (...) You can split the work down. It's all broken down into sections. You just have to make sure you get the right section, everybody agrees. (...) One of the biggest obstacles is if you can separate yourself to find the time to do it.
[Interviewee D]

In some cases, it was highly likely that the users would engage with the source code.

I also find that the code I write must be much clearer in what I'm doing now to what I did before because there will be more people who will actually get to see the code. (...) With the code I'm working on now I have to put enough comments to explain exactly what I'm trying to do. It means that if the user wants to adapt the code to do something different they can understand what it is doing and how they can change it. [Interviewee R]

Another interviewee said that the project he was engaged in was adapting a set of tools to support the production of technical documentation. One of the potential benefits of this approach would be that the users could engage in maintaining the documentation (compare: Quote 5-4).

To summarize, for the vast majority of the participants, the transition from the scientific end-user development to developing for others resulted in a change in the practice of producing user documentation. Before the transition such documentation was almost non-existent whilst in development in the wider context it became essential. Documentation for users could be split into two types of content: 1) the

documentation for end-users with clear instructions on how to use the software; 2) the documentation for users-developers, who may actively engage with the source code.

5.3.2.2 Documentation for oneself

Capturing different kinds of information and ideas about the software for one's own use was also reported in the interviews. This kind of documentation typically included things like algorithmic solutions of problems under consideration, or ideas for implementation, or some software architecture details.

Documentation for oneself had an informal character. Sometimes it was in a form of hand-written notes and sometimes recorded in an electronic format.

I tend to write down more familiar things, more straightforward things. Things that I don't understand, I may put in electronic form because I'd think that I'm more likely to go back and reference it later. [Interviewee M]

The hand-written documentation was stored but was unlikely to be reused.

I have a filing cabinet for written notes and the electronic notes are stored in my computer. The written notes will end up in my filing cabinet. Though, they may well end up not being looked up and then thrown away, a couple of years later when I'm having a clear out. [Interviewee M]

In summary, the production of the documentation for one's own use appeared to be mainly in handwritten form (i.e., on a piece of paper rather than in an electronic format). The scientist-developers tended to store the documents which they wrote for their own use, but they did not always read them again. According to one participant, documentation in an electronic form was more likely to be revisited than hand-written documentation.

5.3.2.3 Documentation shared with other developers in the team

Documentation shared within the development team took different forms. Email was a common, permanent way of capturing information about the software. At the same time a lot of communication between the developers was not captured in any written form.

When there's something that needs to be done, that person will probably usually send an email or just talk. (...) When something wants to be acted upon we all have an email dialog, if it's not supposed to be permanent, if it's just the decisions required. But if it's more permanent, then it will all go on a website [typically in a wiki which the participant showed during the interview] and we like these content management systems. If you have trouble doing something or you want collective input, we've got this as a permanent reference. Also this means that you collaborate. [Interviewee C]

Another interviewee said that a lot of communication within his development team was informal and not necessarily documented in any recorded format. Occasionally, a formal "to-do" list which was discussed

during meetings was documented. The interviewee explained that, since the team was co-located, it was easier to discuss various things related to the software face-to-face than to produce and share any kind of documentation (compare: Quote 5-5).

Summarizing, the documentation that was shared with other developers in the team had different formats. A lot of information about the software was shared within the development team during informal verbal communication that was not captured. However, there seemed to be implicit understanding of what kind of information was “important”, and that information was recorded.

5.3.2.4 Learning how to document

The knowledge and skills for producing software documentation were acquired through an informal self-teaching process. One of the ways to learn how to document was by looking at already existing documentation.

[Q] How did you learn about methods of documentation?

[Participant] Mostly by seeing other examples. I never went on any formal training so I think just by examples, looking at how other people are doing this and thinking this is the way to do it. So if I was using one of these open source libraries and somebody had written for it some good documentation, I thought that's a good way to do it, then I liked the style, the layout. [Interviewee M]

Another approach was by following the style of source code comments which were already present in the program that was under maintenance.

I had no formal training in that at all. I guess the way I learnt was to look at sections that other people had written and try doing it in the same style. [Interviewee O]

Similarly, another interviewee reported following the structure of existing documentation. In his case, he extended a user manual which was still unfinished when he started working on the software. He said that he introduced some changes to the existing documentation but kept a similar structure to the original writing (compare: Quote 5-6).

One of the interviewees discussed in more detail how he developed his documentation skills. He did not believe in any formal training; instead, producing documentation was an almost intuitive process for him.

In creating this documentation and tutorials sometimes I just have a sense how (...) I would write it. But I guess that comes from all the things I've been reading. Mostly it's avoiding what's bad so I'll be reading let's say Python documentation or a Wikipedia article and you think “Why can't I just rephrase it differently? It would be so much easier rather than do it that way, just do it this way.” Or sometimes you say “Oh, that's a neat way of doing this”. So for example this

system was used for the main Python documentation and I thought “Actually this is generated using what? Maybe I could try that too.” So you choose the best aspects but you’re constantly thinking “I didn’t like that style or whatever”. As long as you’re conscious of the mistakes that have been made or you perceive are being made, then it becomes kind of an average of all your experiences. There isn’t a single resource or style guide that I follow. It’s just me. [Interviewee C]

Two interviewees who worked on the same project reported that there was a project coding standards document, which included guidelines on using names and comments in the source code. This document was initiated by one of the interviewees and another colleague whose main role in the project was rewriting and re-engineering a large legacy codebase. The whole project aimed to modernize and restructure a software suite which had been under development for over thirty years. The various components of the software had been written by scientists working in different research institutions. As there had been no comprehensive methodology or guidelines for the developers, they approached the development in various ways and added code which varied in structure and quality. With time, the software suite became difficult to use. At the time of conducting this study, the software suite was being re-engineered and modernized (mainly parallelized). One of the first things which the project adapted was the coding style document. As one of the interviewees reported, the guidelines included in this document were mainly taken from its authors’ experience and based on examples from other scientific software.

To summarise, the scientist-developers acquired knowledge about documenting in an informal manner. None reported taking any formal training in producing documentation. In fact, one of the participants stated that he did not believe in the usefulness of any such formal training. As a template for producing documentation of various kinds, the scientist-developers used existing documentation. Sometimes it was documentation taken from software which they used or came across, and in other cases it was existing documentation in projects which they joined.

5.3.2.5 Perceived benefits from producing documentation

Apart from providing information about the software to the users and members of the development team, documentation production was perceived to be beneficial for other reasons. One of the benefits was that documentation writing was an opportunity for the scientist-developers to reflect on the software which they developed. They could, for example, rethink the implementation details. Documentation was compared to teaching. Writing documentation forced thinking in depth about the software and reconsidering justification of all the choices that were made during development (compare: Quote 5-7).

Sometimes (...) when I was writing documentation that maybe I realize these things. With that parallelisation approach - there could have been a different one, a better one, and that occurred to me when I was writing either the presentation

or the documentation. It didn't dawn on me until I actually started to trying to explain what I was doing. [Interviewee M]

In summary, the participants' perceptions of the benefits from producing documentation extended beyond communicating information about the software to users, other developers, or oneself. The scientist-developers perceived documentation production as an opportunity to reflect on the software they developed and to reconsider implementation decisions.

5.3.3 Documentation production and the user community

The transition from scientific end-user development to developing for others meant that there was potentially a large user community with diverse needs and skills. The interviewees discussed the characteristics of these communities and their needs and expectations. In general the interviewees were aware of who used their software. However, one of the interviewees reported that he participated in several projects in which he never had any contact with the user community. In fact, he suspected that the software which was developed in these projects was probably never used. He found that situation disappointing. Other interviewees also reported that they did not have a direct relationship with the users of their software for some projects.

5.3.3.1 Perceptions of the user community

In general the users were perceived to have potentially two approaches to using the software. The first approach was using the software as a "black box"; that is, the users used the software but did not seek to understand in detail how the data was processed. The second approach ("white box") was engaging with the source code, looking at the implementation details, and potentially altering the code. (compare: Quote 5-8).

There was no correlation between the level of scientific knowledge and engagement with the source code. The same scientist-user could be a white-box user of one piece of software and a black-box user of another piece.

A lot of people would argue that this is a high-level scientific technical code which the people using it would be expected to have a high level of scientific knowledge so they should understand some of the things the developers are doing. On the other hand, a very sophisticated academic in the USA developed his own set of codes which he is very interested in but he's using the external-region codes as a black-box. [Interviewee E]

The majority of users in selected disciplines had little interest in any aspects related to computing.

in the case of the chemistry codes, a lot of people using the codes aren't interested in the computing tools as such. They just want to run the codes with certain inputs to get outputs. [Interviewee P]

One participant said that he encountered situations in which his users preferred to talk directly to the scientist-developers rather than to read the user manual provided. Occasionally the users read the manuals and wanted detailed instructions, which would tell them how to proceed in specific cases. The participant had a general impression that the users did not care about the level of granularity and detail in the documentation. (compare: Quote 5-9).

The perception that the users did not pay much attention to documentation was shared by another participant, who worked on developing very complex parallel codes. He said that the key to using the software in an efficient way was to adjust the existing source code.

We tell users this thing about balancing and they will get back to you with some problems in the results and you would realize that that thing is hopelessly balanced. It's very disappointing. It's not the reason why the results are wrong but as an aside you look at different timings in the coding and you realize that they are not in sync at all and the whole thing is running about 10 times slower than it should be. Then you realize that the users haven't taken seriously what you said.
[Interviewee M]

To summarise, the participants perceived that the users of their software could be of two kinds: black-box users who run the software but do not investigate the details of its implementation, and white-box users who actively engage with the source code and sometimes even contribute to its development. For some users, documentation was not important, as they appeared to use it hardly at all. Instead, they preferred to contact the scientist-developers directly and ask for help.

5.3.3.2 Feedback from the user community

Participants reported that sometimes software users were in regular contact with developers, asking different questions.

We constantly have people [users] who want to know how to do some things, found a bug or they think they found a bug but didn't understand the documentation. So we have an active mailing list and an active community [of users]. [Interviewee C]

Software documentation was sometimes a subject of user feedback, too.

Our users will tell us sometimes if there's something wrong with the documentation - a spelling mistake or something like that. You get a notification. [Interviewee D]

Another participant reported that user feedback that is not directly related to documentation can prompt the developer to look at it.

someone would turn up on a mailing list and say "how do I do this?". Then you would look at the documentation and realize "oh, this is so out of date".
[Interviewee E]

In summary, users' feedback appeared to be a good way of helping the scientist-developers to find various issues in their software or documentation, pointing the scientist-developers directly to missing or out-of-date information. In some cases, users questions about the software prompted the scientist-developers to re-examine and improve the documentation.

5.3.3.3 Documentation and assumptions about users' knowledge

The interviewees said that they assumed a certain level of knowledge related to both the particular scientific domain and general IT skills. For example, users were expected to understand higher mathematics if it was used in the scientific model underpinning the software. Sometimes they were also expected to be able to compile the software's source code on their own machine or to change the software parameters.

I assume they know the physics. If they don't know the physics, they can skip those words. It's a physics experiment, so you have to assume they know physics. [Interviewee I]

These assumptions were reflected in the documentation.

You assume that the person is very knowledgeable scientifically. You don't try to explain what the software does in scientific terms. You assume that if they are trying to use it, they understand what they are trying to do. Then you try and provide a complete and clear description of everything that the user needs to know from the technical point of view. You also assume fairly high level of technical knowledge with respect to how to actually run software on the computer, how to install it. You assume a basic level of familiarity with computer systems which isn't always enough. There would be cases where people need more help. We've never written documentation that assumes a very low level of prior knowledge. Partly it's because the software that we are writing isn't really suitable for people who haven't got that knowledge, because they would probably make mistakes or probably use it incorrectly even if we documented it. Because it's then not so much documenting it as teaching the subject which we obviously can't afford to do. [Interviewee O]

Another interviewee shared a similar view on providing low-level knowledge in the documentation.

So the documentation has to assume a certain level of competency. Either assumes not very much, but it can't get into detail or it assumes you know what you're doing. Otherwise, it ends up being far too long. [Interviewee C]

To summarise, when producing documentation, the scientist-developers gauge the users knowledge, related to both the scientific domain and computing. The information which is captured in documentation is adjusted to match that expected level of knowledge. The users are sometimes assumed to have a relatively basic level of technical skills and knowledge. On the other hand, scientific knowledge is assumed to be at an advanced level.

5.4 Discussion

This section discusses the findings from the analysis of the interviews conducted with the scientist-developers. The study made clear that documentation is produced for different audiences: the developer himself (or his future self), other members of the development team, the users, and user-developers. The developers made different assumptions about these audiences, might prioritise these audiences differently at different times, and might produce different forms of documentation for them. When investigating documentation practices in the two different contexts, that of scientific end-user development and that of developing for a wider community, this study revealed several themes: the community of practice, the benefits of documentation production, the relationship between the scientist-developers and scientist-users, especially the influence of users' feedback on documentation production and the assumptions made about the users' knowledge. The discussion relates the findings to the relevant literature, showing how the themes can be related to other concepts and potentially generalized to wider contexts. Section 5.4.1 discusses documentation practices in the scientific end-user development context. Section 5.4.2 focuses on the role of the community of practice in documentation production for others. Section 5.4.3 considers the benefits that documentation production brings in aiding reasoning about the software development. Section 5.4.4 discusses the relationship between documentation production and the user community. Finally, assumptions about users' knowledge made by the scientist-developers are considered.

5.4.1 Documentation practices in the scientific end-user development context

In scientific end-user development, advancing science is the primary goal (Segal, 2007). This study provides evidence that, in turn, documentation production is subsidiary to software development. In the scientific end-user context, documentation focused mainly on capturing the scientific model underpinning the software, reflecting the focus on achieving the research aim. As one of the interviewees reported, the description of the scientific model was a part of the thesis. In his case, the research aim had to be achieved to obtain the degree. The documentation was given low priority.

It should be noted that the emphasis on documenting the science behind the software is also present beyond the context of the scientific end-user development. Scientist-developers may document the theory underpinning the software if the program is used by themselves or their immediate colleagues (Sanders and Kelly, 2008). Similarly, the study presented in this chapter shows that the scientific model was documented also when the scientist-developers developed scientific software that was intended for others to use.

Other documentation produced in the scientific end-user context not only tended to be limited in vol-

ume, but also was intended only for the end-user developer. For such documentation, the wording used and the contents could potentially be cryptic for anyone else who would try to read this documentation. Even the original scientific end-user developer could find the documentation which he wrote for himself to be difficult to understand over time. Capturing relevant information clearly and accurately enough to allow the scientist (or other scientists) to use the software in the future is a challenge. As one of the interviewees explained, the documentation may express what the developer thought was implemented, while in fact the source code shows something different. If that software has to re-used in the future, the mismatch between the documentation and the source code may lead to problems with using the software and its further development.

5.4.2 The community of practice and documentation production for others

Developing software for use by others included producing documentation for other scientist-developers, including scientist developers within the same development team. Due to lack of clear evidence from the interviews it is difficult to say how the scientist-developers defined what needed to be captured formally and shared within the development team. It is possible that the team members constituted a *community of practice* (Wenger, 2007; Wenger et al., 2002). In such communities, tacit knowledge may develop via shared practice (Wenger, 2007). The team members may, through the practice of working together, have developed a shared understanding of what needed to be documented formally, as well as an implicit understanding of which things had to be captured and shared within the team. Such implicit understanding and negotiation is often present in communities of practice. The findings of this study indicate that not having everything explicitly and formally documented did not affect the team's work. It may be that exhaustive documentation is not needed when there is a well-established community of practice. However, problems may potentially occur if scientist-developers from outside of the community of practice want to work with the software. There may be a lot of valuable knowledge about the software that resides within the community of practice but because that knowledge is never captured it is inaccessible from outsiders.

Brown and Duguid (2002) introduce the concept of "networks of practice" which extends the concept of communities of practice beyond a group of people interacting in person in a close physical location. The exchange of knowledge and experiences happens through various electronic means of communication. As these have been developing increasingly in the recent years, networks of practice have become commonplace. And, indeed, the participants in this study reported that much of their interaction with other development team members happens via email, an internet communicator, a wiki and so on. Sometimes, in projects involving teams located in different research institutions, they hardly ever meet other

members face-to-face. Exchange of information about the software, and production of documentation, happens mainly via the internet.

Consistent with the notion of a community of practice is the observation that scientist-developers sometimes learned how to document by following an example of existing documentation, often an existing piece of documentation in the project which the interviewees joined, but sometimes another piece of software the interviewee was using. The scientist-developers considered what they liked and did not like about the style or the format, and, when they produced their own documentation, they took these considerations into account. Other interviewees picked up the style and the format of existing documentation in the project they joined. In both cases, these learning processes have some common points with the social theory of learning as presented in (Wenger, 2007). The scientist-developers learn how to produce documentation in their everyday work practice, from experience, and from the community of practice they belong to. They learn how to document by using examples of other documentation, by listening to advice from their colleagues, or by analysing feedback from the user community. The knowledge may be passed on directly, from person to person, in discussions, or it may be transferred through an artifact such as an existing piece of documentation. It seems that a lot of learning occurs when it comes to production of scientific software documentation. However, this process is not regarded as explicit learning by the scientist-developers, who commented that “they picked it up as they went along”.

There is some evidence that scientist-developers also learn how to document outside their respective communities of practice. One of the study participants reported that he picked up some documentation styles and formats by reading a Python documentation website¹. It was actually the same participant who said that he enjoyed writing documentation for the scientific software he was co-developing.

Being a part of the community of practice in which scientific software documentation has a low priority may have an impact. For example, if the practice within the community is to capture a particular piece of information about the software in a certain format irrespective of whether it is fit for purpose, it is likely that the members of the community of practice would follow this approach. It may not occur to them that this approach may need improvement, as it is widely accepted and practiced within the community.

5.4.3 Documentation production aids reasoning

In this study, three scientists stated explicitly that documentation production provided them with an important benefit. When they were writing documentation, they had an opportunity to reflect on the design and on the implementation details. Producing documentation prompted some interviewees to think

¹<http://docs.python.org> - this website is created and maintained by the Python Software Foundation.

in depth about the decisions they made and the solutions they used. One of the scientist-participants mentioned that he returned to the code and actually made some changes that occurred to him when he wrote documentation. It should be noted that this experience was echoed by one of the participants of the study on the scientific software documentation crowdsourcing (reported in chapter 7). It is possible that reasoning about the software (triggered by documentation production) may lead to reconsidering the research and even provide new research ideas.

This particular finding about the benefit of documentation production is worth highlighting. As discussed in the chapter covering literature review there is a number of reasons why scientific software documentation production appears to be mainly an overhead for scientists. Highlighting the benefits of scientific software documentation production may help the argument that scientific software documentation merits the investment of scientist-developers' time and effort. When the benefits of producing documentation are made explicit, more scientist-developers may be motivated to write documentation.

5.4.4 The relationship between the user community and documentation production

If the scientist-developers have a close relationship with the software users, feedback from the user community may affect documentation production. Two interviewees who worked in the same project reported that they conducted a survey among their users, asking them for their opinions of the software. The users almost unanimously said that they wanted better documentation. As a result, one of the interviewees spent about six months rewriting the documentation. It should be noted that the project was a mature one and was funded according to a dual-licensing model, in which software was free for academic use but commercial users had to pay for it. In many research projects in which scientific software is developed, it may not be feasible to conduct a similar user survey and then allocate a significant amount of resource to address the issues indicated.

Active user engagement in producing documentation was mentioned by one of the interviewees. He talked about this in the context of introducing the technical infrastructure supporting production of technical documentation. This idea bears some resemblance with the solutions suggested by Berglund and Priestley (2001) who discuss user-driven documentation production (as presented in chapter 3). The ideas presented by Berglund and Priestley (2001) and the indications from this study are further investigated in the next study with the scientist-users (presented in chapter 6). And a case study of crowdsourcing scientific documentation is explored in chapter 7.

Two participants reported that, in a number of cases when the users experienced some issues with the software, it was because they did not read the user documentation. The scientist-developers found this

disappointing. It was not only because the users asked for help and hence added extra work. The scientist-developers were disappointed that, even though they provided the relevant information in the user documentation, it was apparently not read at all or read without proper attention. Reading documentation can be treated as a form of learning, and adult learners are “impatient (...), skip around in manuals (...), rarely read them fully (...) [and] are discouraged, not empowered by large manuals” (Rettig, 1991). Documentation authors need to consider the readers stand point (de Jong and Lentz, 2007). Looking at the text from a reader’s perspective helps to write documents which are actually informative for the users. Aspects such as the style of the document, differences in language and culture between the writers and the readers, differences in prior knowledge or expertise, as well as the perspective on the subject of writing should also be taken into account by the authors. However, it is not clear if, in the scientific software development context, there is enough time and resource to allow documentation practices to be adjusted to follow these suggestions.

5.4.5 Assumptions about users’ knowledge

The scientist-developers expected their users to have a certain level of knowledge, of two kinds: that of the given scientific domain, and that needed for software use and development. For example, one participant assumed that his users were knowledgeable in higher mathematics or physics. Another assumption was that the users had the necessary skill to install the software on their particular platform (which sometimes involved compiling the software source code) and that they were able to alter the source code so that the software became compatible with their input files. The basic assumption about domain knowledge was: if one decided to use for this specialized piece of software, then one must have had a good reason for it. And this reason for using the software involved being a researcher and hence having the relevant background, skills and knowledge in the given scientific discipline.

One of the interviewees commented: “I assume they have the same knowledge as I do”. However, he did not specify what kind of knowledge that was. This approach, of assuming what others know based on one’s own knowledge has been recognized in psychology (Nickerson, 1999). In adjusting one’s message to a particular audience, one’s own knowledge becomes a starting point for assessing the audience’s knowledge. In this study, the scientist-developers producing documentation were familiar with the potential users of their software. The scientist-developers had a background in the same or a related scientific domain, and they used the same or similar research software. Their knowledge, both science and IT-related, enabled them to build the assumptions about the users’ knowledge. These assumptions made scientist-developers omit some low-level knowledge description in the documentation they produced, which helped them to save time. There was also a perception that it was, as one of

the participants commented, someone else's responsibility to teach higher maths or physics required to understand the theory.

What remains unclear is whether the assumptions about the users' knowledge (in any domain) and skills would still hold when the context of software use changes. Sometimes a piece of scientific software developed for one particular context migrates or escapes (Segal and Morris, 2011) into being used in a different context. The preliminary study focused on a change of context from own-use to use by others, but other contexts changes also occur. For example, it is possible to imagine that a piece of software originally developed for one domain of science (for instance, particle physics) may become useful in another domain (for instance, computational chemistry). The level of domain knowledge which may be common for physicists may not be common at all among chemists. As to the IT-related knowledge, an example is reported in the preliminary study, (chapter 2) when one of the participants mentioned that for him the programming lingua franca in his domain was Fortran, but as he started working with younger generations of scientists he noted that many of them did not know this programming language at all. Instead, they knew Java or C++. In that case, the context that changed was the programming culture. In the original context, the software was used (and potentially developed) by scientists who understood Fortran. The context changed to object-oriented programming with the new generation of users-developers.

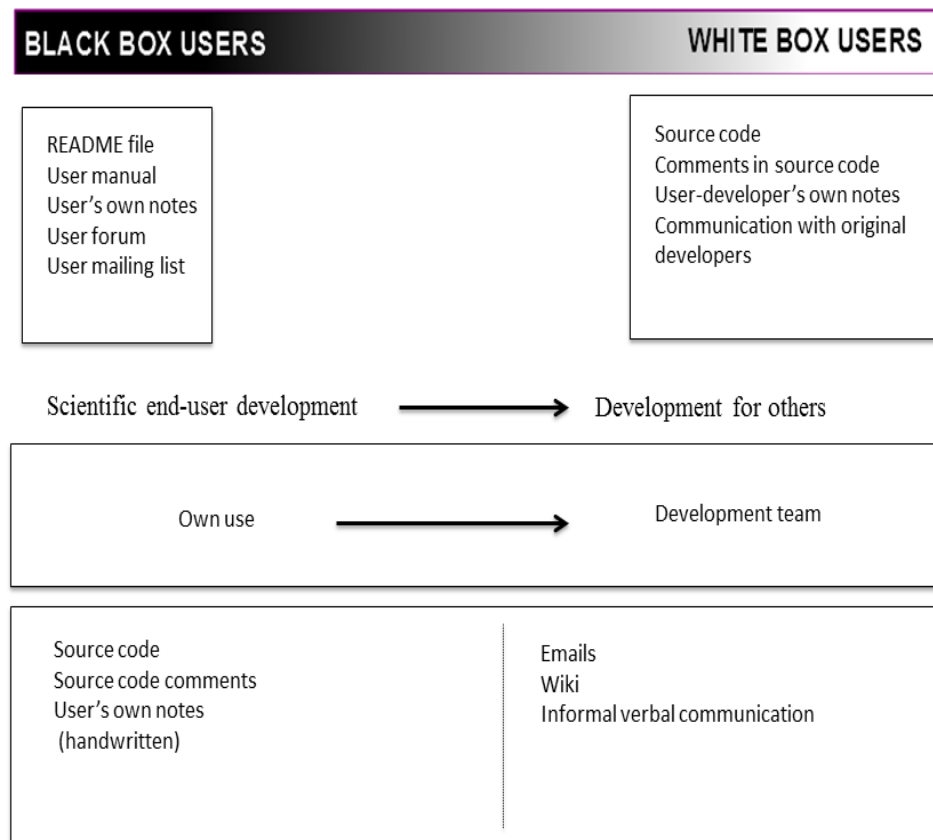
5.5 Summary

This chapter presented findings about documentation production by scientists who develop scientific software. Analysis of the collected data shows that there are significant differences between documentation production in the scientific end-user context, and documentation production in the context of developing software for a wider community of scientists. In the former context, almost no software documentation is produced, as there are very few reasons and motivations for doing so: software is developed and used by one and the same person (or a few close work colleagues); and the software is often a throw-away prototype supporting the achievement of the main research aim. In the latter context, the situation with documentation is more complex. User manuals, in different form and content, are almost always produced. The user community represents a continuum from users using software as only a "black-box" to competent users-developers using the software as a "white-box". Therefore the documentation intended for the users of the software sometimes extends beyond a user manual. A lot of information about the software is shared within the development team in an informal way and is not captured in formal documents. The documentation produced for one's own use may sometimes never be reused. But at the same time its production supports the reasoning process related to software

development. Learning how to document is embedded within a community of practice to which the scientist-developers belong. That is, they acquire knowledge about documentation production by looking at the examples of existing documentation or picking up the approach already present in the team. Feedback from the user community may also influence the scientist-developers' approach to documentation production. The scientist-developers make assumptions about users' knowledge, both related to the given scientific domain and computing skills. It is however unclear how accurate these assumptions are and whether they are amended when the user community changes.

The figure below summarises the conceptual findings and shows how they are related. The continuum from black box users to white box users corresponds with the kinds of documentation which different users (and user-developers) use. The transition from the scientific end-user development context to developing software for others corresponds with the changes of use of software documentation: own use and within the development team.

Figure 5.1: Conceptual findings from Study 1.



Chapter 6 Study 2: scientific software documentation use

6.1 Introduction

This chapter presents the second main study, which focused on the use of documentation by scientific software users and on the documentation they used, in order to identify advantages and disadvantages of different sources from the point of view of the users. Investigating the topic of scientific software documentation from the users' perspective provides information which complements the findings from the first main study reported in chapter 5. The main research question that shaped this study was: *How is scientific software documentation used?*

The research sub-questions were:

- *What are the sources of information/documentation used by scientific software users?*
- *How is documentation used by scientific software users?*
- *What is the role of the user community in scientific software documentation use?*

The structure of this chapter is as follows. The next section presents the methodology used in this study, introducing the participants and describing the data collection and analysis processes briefly. Next, the findings of this study are presented, divided into subsections: 3.1 Sources of software documentation used by end-users; 3.2 Sources of software documentation used by user-developers; 3.3 Perceived disadvantages and advantages of documentation used by scientific software users; 3.4 Documentation provided by the user community. The final section discusses the findings.

6.2 Methodology

More detailed description of the methods used in this study and the justification for selecting them is presented in chapter 4. Similar to the Preliminary Study and Study 1, this study was of a qualitative character to allow for exploration of the topic.

6.2.1 Study participants

Five participants took part in the study. The participants were recruited using the network of contacts established during the recruitment for study 5. It should be noted that recruitment was on purely voluntary basis. The recruitment process was indeed difficult, and there was low response rate to the calls

for participation. Although the total number of participants (5) may not seem large, as the interviews were progressing it became clear that the same themes were emerging. Therefore the decision was made that these five participants provided sufficient data to address the research questions for this study. The criteria for recruiting participants were as follows:

1. the participants were scientists (that is, their main job was conducting scientific research);
2. the participants used scientific software in order to advance their research;
3. at least one of the pieces of software used by the participants was developed by other scientists, rather than a proprietary software package developed by a professional software development company.

There was no specified limit to the number of scientific software packages that the participants were expected to be using. There were also no requirements about the level of expertise and experience with the software packages. The diversity in the used packages, as well as different levels of experience, were highly desirable in order to have a broader view of the use of scientific software documentation.

ID	Domain	Work post	User - developer
A	Theoretical chemistry	Reader - University	No
B	Astrophysics	PhD Student - University	Yes
C	Physics / Quantum physics	Lecturer - University	Yes
D	Particle physics	PhD Student - University	Yes
E	Climate modeling / Dynamical meteorology	Research Scientist - University & National research institution	No

Table 6.1: Study 2 participants

6.2.2 Data collection

The study looked at scientific software documentation from the users' point of view. The research investigated what sources of information about the software the scientists use, as well as the purpose and patterns of use of these resources. The collected data had to inform two areas of research: what people do (their behaviour) and why they do it (their perceptions and opinions). The data collection needed to be grounded in real-life contexts, in order to capture as much as possible about the different variables and aspects which affect the users activities, opinions and decisions (Robson, 2002). Work-based interviews (Holtzblatt and Jones, 1993) allowed the researcher to collect data which met these desired criteria. The participants were

interviewed at their workplace and were asked to perform and/or describe everyday tasks done with the software, in order for the researcher to capture details of their documentation use. With some participants, an additional method, an artifact walkthrough (Raven and Flanders, 1996), was also used. The participants were asked to show how they deal with a specific problematic situation with their software. These methods allowed the participants to recall many details related to these situations and to their documentation use. More detailed discussion of the methods used for this study is presented in section 4.3 in chapter 4.

All interviews were audio recorded, and the audio recordings were transcribed. The length of the interviews varied from approximately 45 minutes to almost 1 hour and 30 minutes. The audio recordings were supported with short field notes taken by the researchers. The field notes included information about the exact website addresses which the participants used or the printed materials at their workplace to which they referred.

6.2.3 Data analysis

The transcriptions of the work-based interviews were analysed using thematic analysis (Boyatzis, 1998). The analysis focused on informing the research questions which underpinned this study. There was a loosely defined list of categories which were assigned to the data in the analysis process. New categories and codes also emerged from the data. The codes were then grouped into themes. The detailed description of the generation of the codes and the themes can be found in section 4.5.1 of chapter 4. The following table shows the clustering of the codes assigned to the themes.

Theme	Codes
Scientific software documentation sources used by end-users	(SOFTWARE-USE)-Sources-Frequency-of-Usage (SOFTWARE-USE)-Documentation-for-Oneself (SOFTWARE-USE)-Sources-Importance-Hierarchy (SOFTWARE-USE)-Sources-Importance-Criteria (SOFTWARE-USE)-Sources-Purpose-of-Use

<p>Scientific software documentation sources used by user-developers</p>	<p>(SOFTWARE-DEVELOPMENT)-Sources-Frequency-of-Usage (SOFTWARE-DEVELOPMENT)-Documentation-for-Oneself (SOFTWARE-DEVELOPMENT)-Sources-Importance-Hierarchy (SOFTWARE-DEVELOPMENT)-Sources-Importance-Criteria (SOFTWARE-DEVELOPMENT)-Sources-Purpose-of-Use</p>
<p>Perceived advantages of documentation used by scientific software users</p>	<p>Documentation-Good-Features (SOFTWARE-USE)-Sources-Forms-Advantages (SOFTWARE-USE)-Sources-Importance-Criteria (SOFTWARE-USE)-Time-to-Find-Answer-in-Source (SOFTWARE-USE)-Time-to-Find-Source (SOFTWARE-DEVELOPMENT)-Sources-Forms-Advantages (SOFTWARE-DEVELOPMENT)-Sources-Importance-Criteria (SOFTWARE-DEVELOPMENT)-Time-to-Find-Answer-in-Source (SOFTWARE-DEVELOPMENT)-Time-to-Find-Source</p>
<p>Perceived disadvantages of documentation used by scientific software users</p>	<p>(SOFTWARE-USE)-Sources-Desired-Changes (SOFTWARE-USE)-Time-to-Find-Answer-in-Source (SOFTWARE-USE)-Time-to-Find-Source (SOFWTARE-USE)-Sources-Issues-Situation-Context (SOFWTARE-USE)-Sources-Issues-Reasons (SOFTWARE-USE)-Costs-of-Bad-Documentation (SOFTWARE-DEVELOPMENT)-Sources-Desired-Changes (SOFTWARE-DEVELOPMENT)-Time-to-Find-Answer-in-Source (SOFTWARE-DEVELOPMENT)-Time-to-Find-Source (SOFTWARE-DEVELOPMENT)-Sources-Issues-Situation-Context (SOFTWARE-DEVELOPMENT)-Sources-Issues-Reasons (SOFTWARE-DEVELOPMENT)-Costs-of-Bad-Documentation</p>

Documentation provided by the user community	(SOFTWARE-USE)-Sources-Issues-Addressing-Tem (SOFTWARE-USE)-Sources-Importance-Hierarchy (SOFTWARE-USE)-Sources-Importance-Criteria (SOFTWARE-USE)-Sources-Purpose-of-Use (SOFTWARE-DEVELOPMENT)-Sources-Issues-Addressing-Tem (SOFTWARE-DEVELOPMENT)-Sources-Importance-Hierarchy (SOFTWARE-DEVELOPMENT)-Sources-Importance-Criteria (SOFTWARE-DEVELOPMENT)-Sources-Purpose-of-Use
--	--

Table 6.2: Codes forming themes in Study 2 - Scientific software documentation use

6.3 Findings

The findings are presented in four main sections. Section 6.3.1 focuses on the use of documentation by the end-users. That is, it presents how documentation is used when the users did not engage with the source code and did not do any development. It should be noted that some of the study participants were end-users in some situations and user-developers in others. Section 6.3.2 presents the findings about how the participants use software documentation in the user-developer role. Section 6.3.3 discusses the the perceived disadvantages and advantages of the documentation used. Finally, documentation produced by the user community and its use are described.

6.3.1 Scientific software documentation sources used by end-users

As the results of the previous study showed, documentation in different scientific software contexts extends beyond a user manual or comments in the source code. The work-based interviews with the scientist-users not only confirmed this but also provided a rich picture of the sources of information about the software.

The definition of scientific software documentation used for the purpose of the study specifies that the documentation is in written form. However, the study presented in this chapter showed that much of the information about the software is not captured in writing, but is shared in conversations. Every participant mentioned this type of knowledge exchange, and it was a valued way of acquiring knowledge. Ignoring the findings about the verbal sources of information about software would potentially limit the understanding of the overall picture of scientific software documentation. Therefore information about

the software shared verbally is discussed in this section.

When scientists used the software as end-users, they usually started by consulting a user manual provided with the software.

There is a massive manual. I have a print-out. So that's something I can look at. (...) For example it falls open here because I can never remember what the correct set of input commands is. So I have to go and check what the correct set of input commands is. It's just so used to being held open at this page that it falls open.
[Interviewee A]

Electronic versions of the manuals had the advantage of being easily searchable.

I keep the manuals stored in my computer. [opens one of them] (...) I don't think I printed this off because it's a large file and it's quite easy to use electronically, you can click [and then the participant simply navigated the document using the interactive table of contents]. [Interviewee B]

Not all user manuals were stored locally by the participants (either on the hard drives of their computers or in a printed version on their desks). Some software packages had manuals available online on their websites. With very few exceptions, online resources were not bookmarked in the participants' browsers. They were accessed by entering the name of the software or familiar keywords into the Google search engine. The practice of accessing favourite online resources related to the scientific software via a search engine rather than by bookmarking them was noted throughout the study. This strategy was particularly useful if the materials available in the preferred online resources were so rich that it was easier to rely on Google indexing rather than to search manually through the whole resource. Google search was also perceived to be very useful in addressing unexpected issues which arose during software use. As one of the scientists summarized it:

If it wasn't for Google we would be lost. We really would. [Interviewee E]

Online resources were used not only to gain knowledge about a piece of software that was already used. The internet was also a resource for finding information about new applications. One of the scientists said that she once found a comprehensive list of different freely-available software packages useful for her domain. She meticulously surveyed all the listed items, checking every link, and trying to estimate if the software was trustworthy (compare: Quote 6-1).

The participant did not explicitly say what made her trust certain websites. However, she mentioned that, for example, she tended to trust websites hosted by research organizations. She explained that she used ISI Web of Science ¹ to search for publications related to a given piece of software.

¹<http://wok.mimas.ac.uk/>

There was this very weird website - when you go on it, it claims to have a super-conductor of a greater temperature. I don't trust these ones at all. I thought that's interesting but then I went to Web of Science and check those in the literature. It gives some explanation but it's not in the literature. So you can't just trust that. I don't trust things when somebody is asking a question and somebody answers it because you don't know who is on to it, they may be wrong. [Interviewee B]

Another participant made extensive use of the online resources which were available from the software's website. The website had a separate section titled "User resources" which included a user's guide, online workshops (as well as materials from face-to-face workshops conducted in the past), links to user forum, and the software helpdesk. This software was freely available and open source. It was developed and maintained by a large research institution, which provided documentation and support for both end-users and potential code contributors (user-developers).

Research publications provided another source of information about the software. Typically, the publications addressed the questions about the science underlying the software or its areas of application. These publications could appear in various journals, although, as one of the participants indicated, some journals such as "Computational Physics Communication" are dedicated to reporting computational science. Research papers were particularly useful if the manuals did not include details about configuration and adjustments necessary for different kinds of applications.

Textbooks were very rarely reported to be a source of information for using the software.

I bought a book about Computation Chemistry (...) it gives me some hints about some things once in a while. (...) On the whole there is quite a lot about this on the web. At one point there was this really nice article from somebody's thesis, just what I wanted and it was on the web. [Interviewee B]

The end-users occasionally referred to the source code for information.

What I do now I just go straight to the source code because I know that's where I can find everything. (...) When I first started out it was more a case of using this [the namespace] and the README file and FAQ and all the other stuff. [Interviewee A]

In addition to self-study or reference materials, the participants reported gaining information from attending workshops, typically run by software providers. The workshops provided step-by-step instruction on how to perform particular tasks. The interviewees particularly valued those workshops allocated time for the attendees to try to use software for other tasks not necessarily covered by the workshop syllabus. After attending a comprehensive workshop, almost anyone was able to use the software effectively. Even for software with rich documentation and help-desk support, the workshops were still extremely popular (even though the attendance was a significant time and financial investment).

The workshops mentioned by the interviewees were all run face-to-face which had definite advantages over online courses.

They [online courses] could be effective in teaching the basics but you would lose that element of actually having a person next to you who you could ask your particular questions. [Interviewee B]

The workshops and courses organised for the software users were a good opportunity to understand the scientific model on which the application was based. One of the study participants used a large and well-supported software suite, for which the providers organised workshops for users on a regular basis. The participant explained that the software was rather easy to set up and use, but the scientific model underpinning it as well as the whole application architecture was extremely complex. One of the large scientific software packages provided a help desk service.

They reply quickly. If I had a problem with [software package], I would generally try and figure it out myself. But often you can't do that. So I send an email to [software package] support. [Interviewee E]

The helpdesk service was efficient and fast but reserved for the most hopeless cases in which all other attempts to solve the problem had failed.

I think you don't want to get to a stage when you are emailing them every day. I try to reserve that for things when it's quite difficult to do really. I don't want to email them every time something goes wrong. Otherwise I suspect they might stop replying. If they know that you send them an email when you are really stuck, then I think it's better rather than with every single problem bombarding them "I can't get this to work". [Interviewee E]

Support was also provided by administrators of the computing infrastructure on which the software was set up to run.

There were people running the software who would answer all questions about the software. But also if there was any machine fault, they would be very prompt. They were the people who would run these workshops for the users of the software. I'd email them. [Interviewee B]

Even if the administrator was not a scientist from the discipline, his advice was still helpful, and he was actively supporting the users.

Other scientists, the participants' colleagues, were a source of information about new programs. Sometimes they even provided initial guidance to using it. Yet, as one of the participants commented, getting to grips with the software required actually working with it directly, trying out different options or completing various tasks. A colleague's introduction could not replace that (compare: Quote 6-2).

Sometimes direct support from the original developers of the software was available.

[the software] was written by 1 or 2 people and they wrote the entire framework. It wasn't documented very well. It was a black box and nobody really knew what happened to it or how it worked. Fortunately my supervisor was the person who wrote it all. (...) So I was stuck on a problem "How do I get it to even run?". He would come down and he would be "Blah blah there you go" and I would be "How did you do that?" and him "Oh that is easy you just do that and that is it". I didn't really understand it first. It took me a long, long time to really get up to scratch with it. [Interviewee D]

Information about dealing with unfamiliar data types or formats or with particular processing steps was sometimes sought in examples of software use. In particular, this strategy was used when the manual did not provide sufficient details and information (compare: Quote 6-3).

In summary, the end-users used a variety of sources of information about scientific software. The manuals, if they were available, were the first point of call. The manuals were used to set up and start using the software. Examples of software applications were another way to learn how to run the software. When the information included in the manuals was not sufficient, information about the software was sought among other users. A significant amount of knowledge was passed verbally during discussions with other users and sometimes with the original developers of the software. In the cases when there was user support available, the information from these sources was also sought, but typically only after all other available documentation failed to provide the answer. In order to gain understanding about the scientific model underpinning the software, the end-users typically consulted related research publications. Sometimes, the workshops run by the software providers were an opportunity to learn more about the science behind the software. Searching for information about the software online, using a search engine appeared to be popular. The results of the search were used if they were known to the end-user (for example, if the search linked to a familiar user forum).

6.3.2 Scientific software documentation use by user-developers

This section considers the use of software documentation by user-developers, those scientists who actively engage with the source code, write scripts to run it, develop new functionalities, or fix bugs. This section identifies the sources of information about the software reported by the user-developers, along with how different kinds of sources are used.

User-developers seek information from comments in the source code on a regular basis, as reported by all three participants who were user-developers. Two of them developed software at the time of the study and hence were able to provide a number of examples of source code comments use. One participant reported that there was a significant amount of commenting by the previous developer, which made it possible for him to work with the code.

A different participant who joined an existing development team asked questions directly of his colleagues.

They would say “This is how to use the software, this is what the data looks like, you can do this to look at the data.” So mostly I’ve been sat down and taught. But that’s probably because I’m a developer as well and I have access to these people first hand. [Interviewee D]

Seeking colleagues’ advice, even though very effective, was perceived to be almost a last resort. It was obvious that abusing the colleagues’ willingness to help would create unnecessary tensions, as asking questions would disturb their work.

Similarly to the end-users, the user-developers used documentation to understand the scientific model underlying the software. The source code was sometimes the first place to learn about the scientific model. Either the source code provided sufficient information, or it gave the user-developer enough ideas to search for answers in other sources.

One of the participants had a different approach to understanding the scientific model, starting from the underpinning scientific model, rather than from the code.

For the scientific perspective I’ve more read papers about the models like this one. Certainly you wouldn’t want to do it just by starting at code. That’s the way to find what variables that you need to change are but if you want to look at something that is going through the transformation, you just look at it and you say “Oh what is that?”. Whereas when you go and you look at the paper or the instructions they have got in here somewhere, they are a lot more helpful. [Interviewee C]

In order to understand how to use a particular method, one of the user-developers simply performed an internet search.

I would go to Google and it would be like “Python minidom” and then the actual method that I want. I would go to the main Python website and this is quite useful. I find this one and I have a look at the way they implemented it. So I read up the methods, what the method does, the arguments that it returns and things like that. [Interviewee D]

Another participant said that, in order to understand the source code, she sometimes needed to consult her colleagues who were more experienced in using it. The particular bit of software that she used was very poorly documented. The user manual was incomplete, and the source code was scarcely commented. Additionally, the code was not well structured which made the code itself almost incomprehensible. The participant explained that, after struggling with the software on her own and looking for information in various sources, she realized that the only way she could deal with it was to ask other

user-developers for help.

Consulting the original developers of the software was sometimes a challenge, because it was difficult to identify who these developers were.

At this point this is pretty much every angle of attack I've been through. The best would be to find someone who was involved in the development of the model itself who actually programmed this variable. (...) It may get a bit of doing given how long it takes to find references about these things. [Interviewee A]

The Google search engine was used frequently for troubleshooting and fixing bugs.

I would write some code, run it, when I get an error I would then take the error and put that error into Google and then see what the error was. [Interviewee D]

Another participant, whenever the internet search proved to be ineffective, consulted his more experienced colleagues about the runtime errors. He did not necessarily expect them to provide him with a clear solution, but rather hoped to receive some hints as to how to proceed (compare: Quote 6-4).

One of the participants said that he rarely used textbooks about programming languages. What he usually needed was information about specific features of a given programming language. However, he found that textbooks, even though they provided a plethora of information, usually did not give a clear answer to the particular question which he had in mind.

In summary, the sources of information about the software used by user-developers were almost as varied as the sources used by the end-users. The first point of call was the source code and the comments included in it. However, if these sources did not provide sufficient knowledge, they would consult the original developers (if they were available) or other user-developers (typically colleagues) who were more experienced with the software. However, the user-developers tried not to abuse this source of knowledge and avoided asking too many questions, too often. Similarly to end-users, user-developers referred to research publications for knowledge about the scientific model underpinning the software. Internet searches were used to find answers to specific programming issues.

6.3.3 Perceived advantages and disadvantages of documentation used by scientific software users

When the participants were describing and showing different documentation sources they were also commenting on the advantages and disadvantages of these resources. The advantages were typically related to ease of use and accuracy of the information sought by the user-scientists. The disadvantages were usually related to inaccurate information, missing details or even whole sections in documentation, and difficult access to the information. The particular assessment of merits and shortcomings varied from participant to participant, as each scientist-user had different needs and expectations for

documentation and used different information resources.

The participant who used a large and complex software suite was very positive about the comprehensive documentation which came with the program. The documentation included not only instructions on how to build and run the software but also examples of use (such as running simulations). There was quite a lot of detail about the underlying scientific model, which in the case of this specific software was complex and difficult to comprehend. The explanation of the science behind the application was one of the main advantages of the documentation (compare: Quote 6-5).

Another participant confirmed that information about the scientific model was an advantage.

I think in the physics code you need some physics explanation. It's useful when you are adding things to say what it is you are doing as well. There is no way of doing that literally. Sometimes it may be a reference to a paper, just saying "For this bit look up this paper" it's got all information which you need. I find that very useful. There is no amount of good variable naming that's going to help to put the information on quantum electrodynamics or something like that. It's just beyond that unless you have like a ludicrous variable name which is "this is a variable which explains the quantum field theory blah blah blah and it's according to this equation". [Interviewee C]

It was also advantageous if the provenance of the method or model was provided.

A well-documented code where you can follow the argument through, it's the most important thing. I mean well commented where there is sufficient comments to follow, where the information has come from, which papers it came from, as well. So some references to published papers; perhaps some information about what is expected to happen but mainly details about how the physics works and where they got that information from. Because then you can go back to the equation and translate back into the computer on your own. [Interviewee C]

Attending workshops was perceived as highly valuable.

You get to meet the people who actually wrote the code. They really know what they are talking about. You have got one of these courses. The idea being at the end of it you are sat down in front of the computer and you run the model. They take you through it. You run it, look at some of the output. At the end of that week you are theoretically in a position to go away and run it. [Interviewee E]

It was reported that one of the main advantages of consulting other scientists was that they were in general recommending something they had tried out themselves and considered useful and accurate. A recommendation from a colleague or a peer-scientist was considered reliable and convincing enough to justify using the program.

Prompt replies from the help desk (if one existed) was a definite advantage.

They have this central help desk and they get a huge amount of emails from around the world. It's something like 400 per month. They generally reply to them all. And in detail. They don't just say "Look at the manual" chapter 3. They will genuinely try and help you out with specific queries. [Interviewee E]

Different forms of documentation had various disadvantages. The participants discussed some of the shortcomings in detail. They provided some insight into the reasons why certain aspects of information sources about the software were problematic for them. Sometimes, the participants explained how they would like problems with scientific software documentation to be addressed.

One of the participants worked with a piece of legacy code which had very limited documentation. One of the main disadvantages which she pointed out were the very short and cryptic variable names.

Variable names...again because it's quite an old code a lot of them are called just very short things like "ena". That's great but what is "ena"? "Ena" crops up all over the place when I search for "ena" in all these files, I find three different counters in everyone that all mean different things. But each time it's just local. And "a3all" I have no idea what that is. There is nothing anywhere that would tell me what is the reference to it because there used to be a six character limit to variable names, so usually they are three characters long and you end up with names like "xyz". [Interviewee A]

Some source code comments were almost completely useless.

Some bits of the code (...) have completely unhelpful comments like "Does this work???". I didn't put this comment there. Someone else did and I don't know who and I don't know whether or not it works but I'm just not touching it. [Interviewee A]

The legacy code with which the participant struggled had another disadvantage: comments were not in English.

I've been using it [Google Translate] on the comments in my code. Because they are in German. Quite a lot of them are in German anyway. Some of them are in Italian. [Interviewee A]

Most of the time this method was sufficient and allowed her to grasp the main idea embedded in the comments. However, from time to time Google Translate did not provide clear translation, and she had to consult another scientist in her department whose first language was German.

Paradoxically, scarce or limited software documentation could have some unexpected positive side effects. One of the participants worked with two software packages, X and Y. Software X was well documented and had an active user community and was relatively easy to use. Software Y was poorly documented, and the participant reported spending considerable time trying to figure out how to run the software and to understand the scientific model which underpinned it. She did that mainly by carefully inspecting the source code, putting together bits of information from relevant publications and from

what her more experienced colleagues told her. Thanks to this diligent work, she gained deep insight into software Y and had a number of ideas on how to improve it. In contrast, the ease of use and good documentation for software X meant that she did not have such deep understanding of the science underlying it (compare: Quote 6-6).

Incomplete or missing instructions were a commonly-reported disadvantage of documentation specifically addressed at end-users.

You would follow the guide and it wouldn't tell you...it would be more like a hint rather than actual detailed instruction. You write your program, run it and then you know you missed something but it doesn't say anything in the documentation. Then you call on somebody to help you and they say "The documentation doesn't say this but you have to put in here or something" [Interviewee D]

A different participant found that the manuals which were not written by native English speakers were sometimes not very clear and understandable.

Some of these [manuals] were not exactly interesting English. I found reading this harder to understand. I had to really think about what they were saying. (...) It sometimes just seemed ambiguous to me. I thought then I'd go and have a look at examples. [Interviewee B]

One of the participants encountered difficulties using an online resource which provided a suite of libraries used in a particular scientific discipline. The participant looked there for a piece of code used in a publication whose results he wanted to reproduce. However, even though the online resource seemed quite rich and advertised itself as an open source community effort to increase reuse of scientific software and ensure reproducibility, he was unable to find the part of the code which was crucial for his work.

What wasn't documented or at least what I couldn't find was how to look through the tree, the code tree and find the relevant bits of code which did different things. I just couldn't find them. (...) It looks like you have to go along to a workshop and learn about it that way. There is some Power Point presentations. So it's not got the right sort of documentation to be useful for me. [Interviewee C]

Some research publications did not include any information about the software.

It's always difficult to try and find details on codes and implementations and specially difficult to often find methods. I've had many cases that someone publishes something in [a particular journal] which is a short letter journal so papers are 4 pages. They very rarely publish sufficient details of the methods to reproduce. Obviously they must have the code associated with that...because that's high profile. But writing the longer code-based papers isn't high profile, then the method basically dies. [Interviewee C]

The attempts to reproduce the results published in the particular paper took the participant a long time and were a frustrating process. The outline of the scientific method was described in one of the few

relevant publications. However, the crucial implementation details were not discussed in any of the papers. As a result, the interviewee struggled with writing the source code which would allow him to reproduce the reported results. The participant explained that some details are simply obvious to publication authors and hence they consider them too trivial to discuss in their papers (compare: Quote 6-7).

The lack of detailed documentation of the scientific model resulted in the participant spending several months trying to build the software which would allow him to reproduce the reported results.

I guess I started in February and I was working until November. (...) Something seems very obvious to him and I'm not as bright as him probably and it doesn't seem quite as obvious to me... (...). In the end it was a very simple thing that I missed... [Interviewee C]

The overall practice of reviewing research papers did not allocate time for checking details of the numerical method used or its implementation.

When I review papers and there is a numerical technique in, I would very rarely get into full details because there is no way of checking it. I can check people's maths but it gets much more difficult to check the details. I know that when my papers get reviewed it's the same thing. The reviewers will normally say "This is explained in enough detail." But they will rarely go through the steps, they rely on the fact that you did that. I think the average time people give towards the review of a paper is between 2-4 hours. It's just not long enough to go through this amount of detail. That is the problem in peer-reviewing. It's up to authors to check their stuff as well but there will be things. They may have explained some things more clearly. By the time you get to an appendix, the reviewer doesn't care any more. The peer review is not going to pick anything or say "this could be more clearly explained" because there is a good chance they didn't read this bit in detail. They are interested in results and conclusions, solving a long standing problem. [Interviewee C]

The participant explained that it was not common for authors to publish their source code, which would enable other scientists to easily reproduce the results or see how the particular scientific method was implemented.

I'm just as guilty with that as anybody else. I don't publish codes on the internet partly because I don't want people reading it and finding bugs and going "This paper is wrong because I found a bug in line 261" which would be good to have these things out there for people to [...] with but at the same time you've got some sloppy piece of scientific code and you really don't want to be sifted over by half of the community. So I guess that's why we all keep these things closed as well which is not as healthy as it ought to be. Open source is the way forward really. [Interviewee C]

One of the participants described how he struggled with reproducing results reported in a scientific publication that reported some significant new findings. The original code used to produce the results was

not available, and the details of the numerical solutions were not discussed in the paper either. The participant said he spent a number of weeks writing the code that would produce the same results. He was completely stuck. The only solution was to contact the original developer, that is the scientist who authored the paper and whom the participant had previously met in person. However, contacting another scientist from another research organization could have been interpreted as a willingness to collaborate. Yet that other scientist was, not only a fellow researcher, but also a competitor. Talking to that other scientist was a potential threat: it could motivate the other scientist to, for example, finish a new paper and get some new results faster than the participant (compare: Quote 6-8).

In summary, participants identified a variety of advantages and disadvantages of different sources of information. The user manuals were a good place to start working with the software and served as good reference materials. However, if they were incomplete or inaccurate, the users struggled with the most basic tasks. Similarly, the source code allowed the scientists to learn about the nuts and bolts of the software, but if it was badly structured and poorly commented it could pose problems. All participants who mentioned workshops were very positive about them, reporting only advantages. There was one significant disadvantage with consulting other scientists. By asking questions, the researchers could unwillingly reveal information to another scientist who could be a competitor in the research world.

6.3.4 Documentation provided by the user community

The interviews showed that consulting other users of the software is a relatively common approach for gaining information about the program, and that the user community is a significant knowledge resource for the software users. The information exchange happened sometimes during meetings in person and sometimes via online channels. User-generated documentation took various forms: mailing lists, internet fora, and so on. Publications authored by other users could also be considered a form of user-generated documentation.

Information about the software shared orally was considered important and valued. The knowledge exchanged within the user community was sometimes not captured. However, this knowledge exchange seemed ubiquitous. Understanding this mechanism is helpful to understand the role of documentation in scientific software development. For that reason the findings about information about software shared orally are presented in this section.

Even with the variety of different sources of information about the software, user-generated documentation was perceived as valuable. User-generated documentation takes different forms. One of them is a user mailing list where the problems were posted and answered by the members of the community. One participant used one such resource quite frequently and referred to the mailing list repeatedly during the

course of the interview.

A different interviewee also checked the mailing list for information about the software. He read the messages posted by other users but never asked questions himself. As he explained later, the information that was already in the mailing list, generated by previous questions and answers, was usually sufficient to enable him to solve particular problems with the software.

I look at the previous queries if I get stuck with something. That's usually via Google. For example if [the software] crashes and gives me an error message, I would type that into Google with quotations around it and they would almost invariably end up back in one of this kind of things [the user forum]. I wouldn't necessarily scroll through all these things [on the forum website] and try to find something relevant. You would generally type the error into Google. It would usually bring you back to this page here. [Interviewee E]

The user forum was highly valued for its accuracy and content.

It's pretty good [the information in the user forum]. If it doesn't solve it here, I think it's probably not achievable. [Interviewee E]

The user forum leveraged the potential of the knowledge about the software residing within the community.

The point of doing it on the forum is that it's there for other people to see. If you do post on the forum, other people can see your problem and if you have the same problem as them, you can see what they did. It's more of an interactive thing. [Interviewee E]

Not all participants found documentation generated by the user community as useful as in the example described above. One of the participants considered mailing lists to be rather useless. She subscribed to several of them. In one case, the subscription was obligatory for all software users. The contents of the mailing list had never helped her to answer any of the questions she had. (compare: Quote 6-9).

For one of the user-developers, user-generated documentation was useful in implementing new functionalities of the software. Other scientists were likely to explore related areas and sometimes shared their work on online fora.

If it still doesn't work, I will then look up examples. People often have forums where they ask questions and they do things which are similar so I do it exactly the same way. I see how other people have done it and try to understand what is going on and play around with my code. [Interviewee D]

User generated documentation was produced by an informal way. There was no predefined list of topic or questions which would be covered. However, this did not necessarily mean that this kind of documentation was messy and chaotic. The content in a rather natural way was organised around topics (for

example, under forum threads). Some fora or mailing lists were more active at particular periods related to what was going on with the software.

Not only software users were active on a user forum (or a mailing list). In some cases the original developers of the software were also generating some content, typically by answering the questions asked by the users. Sometimes the original developers' advice complemented the information provided in the manuals which they provided themselves.

As far as I know for [the software package] this is the place to go because these people have developed it and they keep developing it and keep all the documentation. Pretty much everything about [the software package] is here. This is the port of call, if the manual and README file and peeking in the source code won't do it, this is where you go. Which is great because then you have people like him, just on here answering all these questions. [Interviewee A]

Knowledge exchange and sharing experiences was taking place also during user meetings. These meetings were a good opportunity for informal discussions, especially if other form of communication was not popular in the community.

There are user [of the computing resources] meetings. You don't usually email the other users. I tried to attend those because they are free and it's a chance to meet people in my field without having to charge the university for the conference fees. [Interviewee B]

There were several things which the scientists could learn by attending the software user meetings.

They have several sections [during the user meetings]. One is about how the service is doing and it includes new software they bought [to be run on the computing resource used by the users]. People suggest new software that they may buy. There is a report from those who maintain the service about what goes on there. And then there are talks and presentations, and poster sessions. So you can see if someone is working in the field and ask for advice how they did this. It's really very useful. [Interviewee B]

One of the participants remembered an occasion at which the advice he received during one of the user meetings solved some of the issues he had with the software. It was an equal information exchange in which both the participant and the people he asked received the information they needed.

The information exchange about scientific software at research conferences was a bit more random than during the user meetings. But the conference venues were still a place where scientists could learn about a new software or the (potentially novel) ways it was used. Since the researchers who used and recommended the software were present at the venue, it was also an opportunity to learn what the software

could be used for.

Another participant's concerns related to consulting the user community were related to her reluctance to reveal her lack of knowledge. She thought that asking questions which might have seemed to the rest of the community as trivial would discredit her as a scientist. Her shyness was also another prohibiting factor.

One of the participants shortly but neatly summarized why he consulted the user community:

I think I ask people to fill in the gaps that I found in the manuals. [Interviewee B]

6.4 Discussion

The findings of this study provided a very rich picture of scientific software documentation from the software users' viewpoint. The discussion draws on the selected themes which are not only the most prominent but also on those that relate to the outcomes of the previous studies as well as help to guide the next empirical study reported in chapter 7. The discussion starts with looking into the variety of documentation sources used by the scientists. What was only indicated in the previous study is even more emphasised by the findings of this study: scientific software documentation extends way beyond user manuals. Different sources fit different purposes and are used in different contexts. The disadvantages and problems with these sources are discussed next. One of the issues, the trustworthiness of the documentation is looked at in greater depth. The contexts in which trustworthiness were reported to be key documentation features are considered. The criteria for trustworthiness are also discussed. Then the discussion moves on to the user community as a source of software documentation. Similarly to the scientist-developers, the users tend to form communities and network of practice. The knowledge sharing and experience exchange within these communities appears to be invaluable and, in some circumstances, the participants reported relying heavily on this community generated documentation. Finally, the discussion focuses on the issues related to consulting the user community or anyone else relevant (such as the computer infrastructure providers) with regards to the software.

6.4.1 The rich variety of documentation sources

Participants used a variety of sources of information about the software and also about the research done with the particular software. Both the end-users and user-developers referred to many different kinds of information sources. There is evidence that scientific software documentation extends well beyond a user manual or comments in the source code.

On one hand the available documentation, such as manuals, was often insufficient for the users to use the software effectively. Information was missing, inaccurate, or even written in a foreign language. In all

these cases, it was understandable that the users sought alternative documentation. On the other hand, even when the manuals were essential and effective at the early stage of using the software when the users needed the basic knowledge to get the software up and running, the users' more advanced queries about the software extended beyond the material covered by a user manual. The findings show that, in the later stage, when the users successfully built the software or installed it from binaries and used it several times, the manuals were used as reference materials only. The users sometimes needed to look up the parameters or the workflows but in general the manuals were put aside.

This may explain why some participants did not have the manuals handy in their everyday workspace. It does not mean that there is no need to produce comprehensive manuals. These documents are crucial for the users to start working with the software. The way the scientist-users use manuals appears to be similar to the way professional developers use high-level technical documentation, for example, documents describing software architecture (de Souza et al., 2005). The scientist-users needed the manuals usually to make their first steps with the software: to understand how to set it up and start working with it. As they worked further with the software they needed more detailed information. For example, they required information about a specific software configuration or wanted to know how to run a particular experiment or a simulation, and so on. Then they sought other documentation: from research publications or comments in the source code to information on users' mailing lists and in the community.

Participants regularly shared information about software orally, and this sharing was very important. Even though this information was never captured and hence does not adhere to the definition of documentation used for the purpose of this thesis, it is discussed in this section because of its importance.

The study reveals that hands-on workshops were popular among the scientists. They gave them the opportunity to try to run some of their own tasks in a safe environment. There was a software expert available on site who was competent enough to help with tasks more sophisticated than the standard exercises presented in the workshop. Workshops offered something extra in comparison to manuals, which were used for independent study. Workshops and workshops were structured, had set goals, and allowed the participants to interact with the instructors or other participants.

The scientist-users switched seamlessly between different sources of information. The user-developers displayed a similar behaviour. The answers to specific questions built up a bigger picture about the software. It is possible that, since every scientist-user had a different set of questions, the knowledge which each one of them constructed about the software was unique. At the same time, there was a certain common core knowledge that the users shared, and, thanks to that, they were able to communicate about the software and exchange their experiences, even though they might have learnt about the application

via different sources and experiences.

6.4.2 The main problems with documentation use

The problems with documentation that the scientist-users experience are not exclusive to the scientific software development context. Incomplete or erroneous software documentation has been a longstanding issue more generally. Studying the problems with documentation use in the context of scientific software may help to find solutions applicable for this community rather than applying generic solutions.

Incomplete user manuals made it extremely difficult for users to get the software up and running. The brief instructions without sufficient details indicate that those who authored the documentation either did not have enough time to write documentation or assumed that some things would be obvious to the users. The missing information was sometimes just a short instruction. Those more experienced with the software were able to address the issue almost immediately with quick advice, suggesting that the missing information was not necessarily something which would require much time and effort to add. It is not surprising that there was not enough resource allocated for writing documentation in the cases in which scientific software was originally developed in an end-user context and was not originally intended for use by a wider community. Moreover, because software is rarely rewarded directly with academic credit (Howison and Herbsleb, 2011), the resources for its development are limited.

It might be that, in the cases in which documentation was not complete and did not provide answers to the users questions, the software had actually escaped or migrated from the scientific end-user context into wider use (Segal and Morris, 2011). If the original developers did not anticipate that anyone except from themselves or maybe a few lab colleagues would (re)develop the software, they would likely have taken shortcuts with documentation. For example, both this and the previous chapter report examples of comments written in different languages: one of the user-developers said she had to deal with the code in which comments were written in German (which she did not know), and one of the scientist-developers in the previous chapter said that she used to write comments in her own source code in Spanish, her mother tongue. In both cases, the originators created the code for their own use, and hence commented the source code in the language that was the most convenient for themselves. In the cases when the software is used outside of its original context, usability of the software may be an issue Segal and Morris (2011). And indeed the findings of this study support that showing that incomplete or inaccurate documentation affects ease of use. Some user-developers struggled as they tried to understand the source code in order to alter it to suit their own purposes.

Another factor is that scientific software development itself does not benefit the career of a scientific

researcher (Howison and Herbsleb, 2011), and scientists often perceive software development as an easy thing to do (Segal, 2005a). This may be a reason why software documentation is not given much attention during the review process. The peer review process for the publications which were produced using a particular piece of software did not ensure that information about the software was sufficient or even included in the paper. As one of the participants explained, the reviewers did not have time to thoroughly check the underlying mathematical model described in the paper. They simply trusted the authors that it was correct. It may be assumed that since there was no feedback about the scientific model or any implementation details (if they were included in the paper), the authors did not focus their efforts on making this information entirely clear.

Another issue is the trustworthiness of the sources of information. Although it was explicitly raised and discussed in detail by only one of the participants, it was clear from what she said that it was a problematic area. As other research shows, evaluating trustworthiness of the scientific software remains a challenge (Hook and Kelly, 2009). Therefore the topic closely related to that problem, of trustworthiness of the sources of information about the software, is discussed separately in the next section.

Insufficient information about the software included in the published papers leads then to major problems with reproducible research. Issues with reproducibility in computational sciences have been one of the major concerns in the scientific world. In their guest editors' introduction to the special issue of "Computing in Science & Engineering", Fomel and Claerbout (2009) show that problems with reproducibility have accompanied computational science for many years now. There are already journals, among them "Science" and "Nature" (Vandewalle et al., 2009), which require the authors to make available their data as well as their methods and tools (including source code). Certainly, this could help to address some problems of the scientists who try to reproduce the results reported in the publications. However, if the submitted source code is not documented, or if that documentation is not peer-reviewed (like the paper is), then many problems with documentation still remain unsolved. Of all the problems concerning scientific software documentation use, the issues related to reproducibility may be the ones that are exclusive to the scientific software context.

6.4.3 Trustworthiness of documentation resources

Even though the participants reported using a large range of sources of information about the software, the matter of trustworthiness of these software resources was not always discussed. At the same time, there was an indication that there were issues with trustworthiness of some of the sources. In particular, the information available on the internet required some verification. Only one participant explicitly described how she estimated if the resources were reputable and well established. Howison and Herbsleb (2011) found something similar: scientists stated that they considered certain software packages reliable

because they came from reputable sources.

It may be that the trustworthiness of different sources of information about the software was implicit knowledge. That is, it may be that there is a common understanding among user-scientists, or scientists in general, with regard to which sources are trustworthy and which are not. As one of the scientist-users mentioned, the advantage of consulting other users was that they either recommended specific software packages or explained how to use them and how to troubleshoot problems. Acquiring the understanding of what is trustworthy may be an element of the process of becoming a member of the user (and scientific) community - a community of practice. Trust in particular sources of information is a part of everyday practice, and scientists simply acquire it as they go along. Again, this echoes the findings noted by Howison and Herbsleb (2011). The question which arises here is whether there is a common understanding of what is reputable and established in the scientific world.

The participants of this study consulted other scientists about the software and found their advice trustworthy. This trust may extend beyond occasional advice about running the application. Peers' recommendation may be a very strong reason for adopting software (Joppa et al., 2013). While this may be a good indicator as to whether a piece of software is indeed useful, it may not be sufficient. Joppa et al. (2013) note that scientists rely on others' opinions to mitigate the risk that they would themselves pick something that does not perform the work that they need or, even worse, generates incorrect results. On the other hand, scientists using a piece of software recommended by "reputable peers" may not really understand how the software works and hence may have difficulties assessing if it is really relevant for their research. The community of practice is a great asset, but it should be treated with a degree of caution and critical appraisal.

Interestingly, in the cases in which the user-developers sought information that helped them in dealing with technically-oriented tasks, they were quite happy to simply check out internet search engine results. It was a quick and easy way to find answers to the questions about writing source code and solving programmatic problems. Trustworthiness of the sources which provided the answers did not seem to be considered at all, as "the proof was in the pudding": the code would either compile and run or not. It is unclear whether these solutions were effective in long term. It is possible that, in order to assess the trustworthiness of sources of information related to technical aspects, advanced knowledge in software development knowledge is necessary. Not all scientists have this knowledge (for example, Vidger et al. 2008). Providing scientists with support in assessing the trustworthiness of the sources of information may be worth considering.

6.4.4 Documentation of the scientific model and details of its implementation

Documentation of the scientific model and method underlying the software is crucial for all users, both end-users and user-developers. For both of these groups, it is important to understand what the software is supposed to do and on what scientific basis the whole program is built. It is also important that the limitations of the model are articulated. This knowledge is essential if they want to progress their own research and publish. For the user-developers there is an additional aspect: only after understanding thoroughly the science underpinning the software can they build new modules and add new functionalities.

The matter of documentation of the scientific model underlying the software together with the details of its implementation becomes even more crucial when the scientists want to reproduce the research reported in a published paper. Often, the actual software is not available. Therefore the scientists try to recreate it based on the published information. They could be then considered a special case of user-developers. Their goal may be similar to those of user-developers who work on an available copy of scientific software. Reproducing the results reported in the paper may be an interim step toward developing their own source code, possibly extending the reproduced code. Insufficient information about the software which these scientists want to recreate results in them wasting valuable time and gradually becoming incredibly frustrated. As discussed earlier, the peer review process and the academic publishers' requirements do not emphasise providing enough detail about the computational model, algorithm and implementation details which would allow other scientists to recreate the original source code. Introducing a common policy of requiring all authors to make the code (and data) available in established repositories is certainly a good idea. However, the findings from this study show clearly that the source code alone is typically not enough. Reproducible research depends not only on the source code being available, but also on the relevant documentation.

An additional incentive to make one's research reproducible by making a copy of the source code available online and providing enough documentation, is that this factor may contribute to the papers produced using this software being highly cited (Vandewalle et al., 2009). And academic reputation grows as the number of citations grows (Howison and Herbsleb, 2011).

It may seem that, since consulting the scientific community about software appears to be common, any scientist who is stuck trying to reproduce another scientist's work could simply contact the original author and ask for help. In many cases, this approach could be ideal. But there may be several potential risks and problems. For example, the original authors may be not contactable or may refuse to provide a copy of their source code (or a data set) (Savage and Vickers, 2009). Another problem, discussed earlier

in this section, is that, by asking questions, the scientists may unwillingly reveal information to their potential competitors.

6.4.5 User community-generated documentation

Throughout this study, the user community was portrayed as a rich and useful information resource. Other groups, such as IT infrastructure administrators or the original software developers, were also consulted. However, other software users, other scientists, were reported to be a source of information by every participant of this study.

Knowledge about software accumulates from a rich variety of users' experiences using it for research. In the scientific, or maybe rather the academic world, this exchange of experiences and knowledge sharing is enhanced by the overall culture of telling others about one's work. It is a part of the job of a scientist to communicate with his community, both at software user meetings and research conferences. Mailing lists and user internet fora are another virtual space for knowledge and information exchange. These communication spaces can be diverse and of a very international character. But, as this study shows, echoing the findings reported by Lanier (2011), this factor does not prevent effective communication.

In an ideal world, consulting the community would solve all problems that scientists have with software. But, of course, a variety of issues are present. First, asking questions may be risky. The enquirer may feel that he may lose his reputation within the community by asking trivial questions and exhibiting a lack of knowledge in the areas which are considered obvious. Personal traits such as shyness may prevent a scientist from asking questions, even those difficult ones which could possibly be appreciated in the community as a trigger for an interesting scientific discussion. As was discussed before, asking questions means also revealing information about one's own work. In the competitive research environment governed by the rule "publish or perish" (and importantly, be the first one who publishes) revealing too much about one's work may have unwanted effects.

Yet, despite these potential risks, the user community remains invaluable as an information source about the software. Scientist-users form communities (Wenger, 2007) and networks (Brown and Duguid, 2002) of practice. They work in research institutions (often academic ones), so they can understand the cultural context: building reputation through research publications (Howison and Herbsleb, 2011), the dominating perception that software development is easy and can be done by anyone (Segal, 2005b), the constraints around work organisation which make some software development methodologies used widely in the industry unfit for the scientific software context (for example, Wood and Kleb 2003), and so on. A community of scientists, both users and developers, clustering around a piece of scientific software has more in common than just using the same piece of software. This common background may

be a factor which increases understanding among the user community. This may also be a reason why the members of the community trust each other. The study showed that the scientists were eager to try out a new piece of software if it was recommended by other scientists. This behaviour is consistent with Diffusion Theory (Rogers, 1962). Diffusion is “the process by which an innovation is communicated through certain channels over time among the members of a social system” (Rogers, 1962, p. 10). The “innovation” in the cases discussed in this dissertation can be a piece of scientific software or a matter related to software development (for example, a programming language). The “channels” can be any of the virtual platforms, such as a mailing list, or a face-to-face scientific meeting. The “social system” is a scientific community. Tools, methodologies or approaches are adopted because other members of the community adopt them and recommend them. If the common scientific background is combined with active help from the user community, it may even more easily attract new software users (Bagozzi and Dholakia, 2006). The study of scientific software documentation use shows that both the novice scientist-users and the more experienced ones participate in the user communities. It may be that, if the latter adopt and recommend the “innovations”, the diffusion of innovation is even faster and more powerful.

The findings of this study show clearly that the user community can be a great information resource about scientific software, filling in the gaps in the documentation. The community can provide answers to questions which arise as science progresses and the scientists want to apply the software in new areas. The question is whether there is a way to harness and deploy this potential of the user communities to generate (dynamic) documentation. The next chapter 7 addresses these questions by investigating crowd-sourcing documentation in an open source scientific software project.

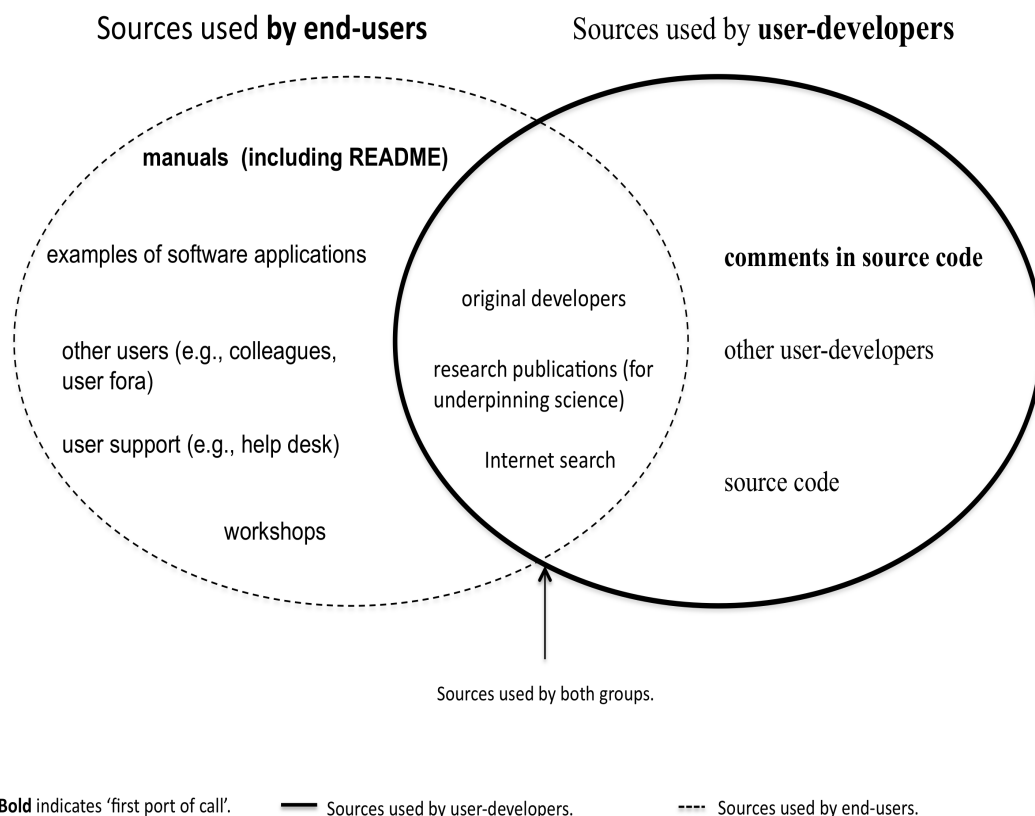
6.5 Summary

This chapter presented and discussed the findings of the second main empirical study, which looked at scientific software documentation from the point of view of the scientist-users. Thanks to the method of data collection used, the work-based interviews, the study captured many significant details which allowed the researcher to have better insight into the use of documentation and, in general, its role in scientific software. One of the outcomes of this study is that documentation in scientific software extends beyond user manuals or comments in the source code. As the users need different information about the software at different stages of their work, they move among a variety of sources of information about scientific software. Clear information about the scientific model underpinning the software was perceived to be a necessity. Workshops during which the scientist-users had an opportunity to perform tasks and also try to run the software for their own problems were valued highly, especially because they provided the opportunity for the users to obtain interactive and direct help and support. One of the

crucial issues related to scientific software documentation was the one concerning reproducibility of the research. Inadequate information about the software appears to be one of the main factors hindering reproducibility, which is a key element of scientific research. Trustworthiness emerged as a concern, although it was not often discussed explicitly by the study participants. It is possible that trustworthiness of resources is something understood implicitly and negotiated within the community of practice to which the scientist-users belong. Finally, this study provided strong evidence that the user community produces significant amounts of useful and highly-valued documentation. Consulting other users happens almost all the time and has many advantages, from filling in gaps in the documentation provided by the original developers, to learning about new ideas and approaches to research.

The figure below summarises the conceptual findings and shows how they are related.

Figure 6.1: Conceptual findings from Study 2.



Chapter 7 Study 3: crowdsourcing documentation production

7.1 Introduction

This chapter presents a case study of the SciPy and NumPy Documentation Project. SciPy is “open-source software for mathematics, science, and engineering. It is also the name of a very popular conference on scientific programming with Python.” (www.scipy.org). In fact, SciPy is becoming an umbrella term used to refer to a wider set of scientific software tools written in Python and depending on the core SciPy library. This term is also used to refer to the community of developers and users clustered around these tools. The SciPy software depends on the NumPy library which is a “fundamental library needed for scientific computing with Python” (www.scipy.org/more_about_SciPy).

The SciPy and NumPy Documentation Project is an example of documentation crowdsourcing, in this case documenting the SciPy and NumPy scientific software packages. The focus of this particular study was shaped by the analysis of the previous studies, which showed that documentation is one of the main issues in scientific software development. Scientist-developers typically have limited resources and time to develop software, let alone to produce documentation. On one hand, scientific software is often developed by a single developer for one-off use; on the other, scientific software that is used over time tends to evolve and change. Therefore the documentation needs to be dynamic to keep up with the software changes. Given their focus and resources, producing and updating documentation is a major challenge for scientist-developers.

On the other hand, as Study 2 showed, in cases when documentation provided by the scientist-developers is not sufficient, the software users turn to sources of information created by the rest of the user community. A significant and very useful body of knowledge about scientific software is created by its users. This creation is an organic and a bottom-up process. Hence the product, the generated body of knowledge, may be often somewhat chaotic and disorganized. Nevertheless, it is an invaluable resource which helps the scientist-users to fill in the gaps in the documentation provided by the scientific software developers.

All these outcomes lead to questions: How can this potential of the scientific software user commu-

nity to build a body of knowledge about the software be used? Is it possible to harness this organic process of documentation generation in order to make the output more organised? Lakhani and von Hippel (2003) derive similar questions from their study of open-source software user-to-user assistance. They say that “it is important to analyze the micro-level functioning of successful open source projects to really understand how and why they work” (Lakhani and von Hippel, 2003, p. 940). They suggest that research should look into (among many topics) “how is coordination achieved among open source software contributors; how can problems be segmented into module of a size that fit the sources and incentives of individual users to effectively contribute?” (Lakhani and von Hippel, 2003, p. 940). These questions could be applied not only to source code contribution but also to providing information about the software and producing documentation. It should be noted that many scientific software packages are open source.

As an example of crowdsourced documentation, the SciPy Documentation Project provided an ideal opportunity to explore the topic. This study investigates the process of documentation crowdsourcing in order to understand how it works. The research questions for this particular empirical study were:

- *What may lead to crowdsourcing scientific software documentation?*
- *How can crowdsourcing scientific software documentation be organised?*
- *What are the benefits of scientific software documentation crowdsourcing?*
- *What are the challenges of scientific software documentation crowdsourcing?*

As Doan et al. (2011) discuss, there are multiple definitions of *crowdsourcing*, e.g., Obtain (information or input into a particular task or project) by enlisting the services of a number of people, either paid or unpaid, typically via the Internet¹. Scholars, popular writers and journalists attempt to define the phenomenon. In their survey of crowdsourcing systems Doan et al. (2011) propose the following comprehensive definition:

“we view crowdsourcing as a general-purpose problem-solving method. We say that a system is a crowdsourcing system if it enlists a crowd of humans to help solve a problem defined by the system owners and if in doing so, it addresses the following four challenges: How to recruit and retain users? What contributions can users make? How to combine user contributions to solve the target problem? How to evaluate users and their contributions?” (Doan et al., 2011, p. 87-88)

The above definition does not explicitly say if the “system owners” are different from the “users”. It should be noted that in this study some owners were also users. That is, some of those who set up or

¹Oxford Dictionary, <http://www.oxforddictionaries.com/definition/english/crowdsourcing>

co-organised the process of documentation crowdsourcing were contributing efforts themselves.

This chapter is structured as follows. The case study is introduced and described. Then, the data collection and analysis are discussed. Findings with the relevant evidence are then presented, followed by the discussion of the findings. The chapter is closed with a summary.

7.2 Case study introduction

This section introduces the case study, using information drawn mainly from two progress reports (Harrington, 2008; Harrington and Goldsmith, 2009) and one technical overview (Van der Walt, 2008) published in the SciPy Conference proceedings. In order to keep a clear picture only selected and relevant facts are presented in the paragraphs below. The interviews and the data from the mailing lists archives confirmed the reported facts.

NumPy development started around 1998, and work on SciPy followed it soon. One of the challenges which the users of these packages were facing was insufficient documentation. SciPy and NumPy were partially documented and some users were creating their own, also partial, documentation. During the discussions (conducted mainly on the NumPy-discussion and SciPy-Dev mailing lists), the idea of engaging the user community in producing documentation emerged. A wiki was set up to enable community members to register and add documentation. Eventually the community effort focused on improving NumPy and then SciPy docstrings. Docstrings are string constants (which start and end with a marker `"""`) which are defined within a Python object's definition. Docstrings can be accessed easily in both the Python interpreter and from Python programs via the `__doc__` attribute or by calling `help(object)` (the latter method is provided by the `pydoc` module). The intention is that docstrings describe the object's most important properties. Not only can they be accessed interactively, but they also make it possible to build documentation automatically from the object source code.

Originally the docstrings could only be edited by accessing the source code. The code was kept in an SVN repository (N.B.: (It is now hosted on GitHub¹). Only a small group of developers had write access to it. New developers were given write access only after contributing a substantial amount of code (which was reviewed before committing) and proving their ability to handle the code securely. The community wanted to maintain this status quo. Lifting all access restrictions on the repository seemed to be quite risky. It could potentially result in a large number of contributions from inexperienced developers which could lead to introducing bugs. A solution which allowed practically anyone to edit docstrings but at the same time keep the code safe was needed.

¹<https://github.com/numpy> and <https://github.com/scipy>

At roughly the same time as the wiki was created, the developer who started NumPy development set up a “livedoc” site on which a documentation tree was built dynamically from the module docstrings. It gave read-only access to docstrings but was one step closer to a documentation solution.

In the meantime, an initiative called “Making Python Attractive to General Scientists” (later known as the Accessible SciPy Project or ASP) was launched by one of the SciPy and NumPy users and one of the developers. Their aim was to expand the SciPy user base by improving documentation (in various forms), enhancing the website and providing better packaging. All of these discussions and initiatives – setting up the wiki, the livedoc site and the ASP initiative – were overlapping in time and were interconnected. In their posts about the formats, standards or the organisation of work, the community members often referred to all of the initiatives.

Eventually, one of the ASP initiators, a SciPy and NumPy user and a university professor secured funding to employ one of the NumPy developers full time to improve NumPy docstrings. As Harrington stated in the first progress report (Harrington, 2008) as well as in the interview, the main reason why he decided that this particular part of the documentation needed improvement was that the main obstacle for new NumPy users was poor reference documentation.

It became clear very early that the developer employed full time to edit the docstrings would be unable to fulfill this task on his own, and that engaging the community was essential. In the meantime the two tools, the wiki and the livedoc site, were combined in a system developed by one of the developers from the community (with assistance from a few others already involved in the wiki setup). As stated in the technical overview (Van der Walt, 2008, p. 27) “The new framework supported all the features of the old one, but, being a web app, allowed us [the documentation project team] to add things easily such as workflow, merge tool, access control, comments and statistics.” One of the key features that the framework provided was “three-way merging”. That is, the docstrings in the online documentation system can be updated with those in the source code repository. Hence, those editing docstrings in the online system are able to work on the most recent versions. A similar update was possible the other way round. A patch could be generated from the documentation editing system and applied on the latest version of the code in the repository. The figure below depicts how this system worked.

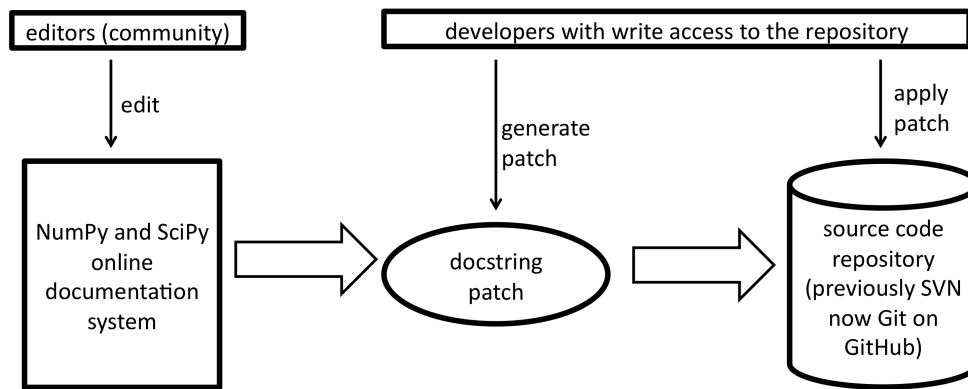


Figure 7.1: NumPy and SciPy documentation system workflow.

The whole documentation effort was launched under the name of the Documentation Marathon in mid 2008. The first full-time documentation editor was hired for six months. A few months later, another full-time editor and coordinator was hired for about 18 months. However, since funding ceased, there has been no full-time person editing the docstrings and managing the community effort.

The above case study overview provides essential background information about the SciPy/ NumPy Documentation Project. More insights and findings are presented, along with the relevant evidence, later in this chapter. It should be noted that, although the documentation system still exists and is in use, the most intensive period of the Documentation Project fell between early 2008 and late 2010, while the coordinators were in post. As a result, it has been possible to study the phenomenon of scientific software crowdsourcing from its beginning to, maybe not its end, but a stabilisation point.

7.3 Methodology

This section discusses the data collection and analysis. In a case study, data is collected from different sources and, typically, using different methods (Runeson et al., 2012). A more detailed discussion of how this study fits into the recommendations for conducting case studies in software engineering is in chapter 4.

7.3.1 Data gathering

The data were collected using multiple methods and sources. The sources were: the archives of three mailing lists, nine semi-structured interviews, two progress reports, one technical overview, the project website (docs.scipy.org) and logs from the server hosting the documentation infrastructure.

The largest source, in terms of volume of data, was the archives of three mailing lists: NumPy-discussion, SciPy-Dev and SciPy-User. All three archives were searched using the Gmane public mailing list archive search engine with the keyword “documentation”. Gmane provides a smart search engine, and hence

all posts which included words stemming from “doc*” (so, for example “documenting”, “docs”, “doc-strings”) were included in the search results. The search was run on archives dated between the beginning of each of the mailing lists until April 2012. Since the main events and discussions related to the documentation crowdsourcing project took place before 2011 or even 2010, it can be safely assumed that all relevant information from the mailing list archives was captured. The mailing lists search returned 2,230 results. All of these were then screened, and 477 posts which were relevant to crowdsourcing documentation were selected for detailed thematic analysis. The thematic analysis was performed using the NVivo software.

The second source was semi-structured interviews conducted with key stakeholders involved in the documentation project, identified in an initial analysis of the mailing lists archives. Nine of them were available for interview. Two were interviewed in person during the SciPy 2012 Conference. The remaining seven were interviewed either via Skype or phone. The interviews lasted between 30 and 90 minutes; most of them took just under one hour. All interviews were audio recorded and then transcribed. Thematic analysis was applied on the transcriptions using NVivo. Further details on the data analysis are presented in the next section.

ID	User/developer	Role
A	Developer	One of the main developers.
B	Developer	One of the main developers and one of the full-time documentation editors.
C	Developer	The main developer of the documentation infrastructure; helped with the organisation and management of the Documentation Project.
D	User	The main organiser of the Documentation Project.
E	User (but with in a lot of software development experience, however not in NumPy/SciPy)	One of the full-time documentation editors.
F	User (now also developer)	One of the most active documentation contributors: helped with setting up the infrastructure and organisation.
G	Developer	One of the main developers; helped with setting up the standards and some organisational matters in the Documentation Project.
H	Developer	The developer who started NumPy development and contributed largely to SciPy development.
I	Developer	One of the main developers; helped in organising the Documentation Project.

Table 7.1: Study 3 participants

The third source of data was publications: progress reports on the SciPy Documentation Project published in the SciPy Conference Proceedings (Harrington, 2008; Harrington and Goldsmith, 2009), a

technical overview paper published in the SciPy Conference Proceedings (Van der Walt, 2008) and the project website (docs.scipy.org). The analysis of this data focused on identifying and selecting the information relevant to the findings from the analysis of the interviews and the mailing lists archives. This information was also the basis for the introduction in the previous section.

The fourth source of data was logs from the server which hosted the infrastructure developed for crowdsourcing documentation. The raw data was in a form of *.csv files. The logs from the server included information such as the number of editor accounts registered on the system, the number of edited words per editor, the total number of edited, proofed and revised words in different time periods and so on. The analysis of this data was quantitative. The results of the analysis provided some descriptive statistics about the SciPy Documentation Project.

7.3.2 Data analysis

The two largest sets of data, the selected posts from the mailing lists archives and the interview transcripts, were analysed using thematic analysis. The process was similar to the data analysis process applied in the previous three studies (see chapter 4). First, the data were coded with one or more codes that emerged during an iterative coding process. As the process of data coding progressed, and new codes emerged, the new codes were applied across the whole data set. The definitions of the codes were revised and updated where necessary. Once all data were labeled, the coding was refined. Similar codes were identified and merged to avoid redundancy. The codes were then grouped to form the themes which are presented in the findings.

The codes which were used to form the themes were:

Theme	Codes
Reasons for crowdsourcing documentation	Arguments-For-Crowdsourcing; Documentation-book; Existing-Documentation; Finding-Developers-for-Code; Issues-w-Documentation; Reasons-for-not-Producing-Documentation; Spontaneously-User-Contributed-Documentation; Suggestion-Documentation-Feedback-Importance; User-Suggestions-Complaints-on-Documentation;

<p>Infrastructure for docstring crowd-sourcing</p>	<p>Arguments-For-Wiki; Arguments-For-Livedocs; Arguments-For-PDF-Documentation; Docbooks; Documentation-Forms; Documenting-Formulae; Existing-Documentation-Conversion; Implementation (Module-Submission-Review-Process); Implementation-Documentation-Website; LaTex-ReST-Word-Tables; LyX-Advantage; Suggestion-Documentation-Live; Suggestion-Documentation-Offline; Suggestion-Documentation-Wiki; User-Suggestion-about-Doc-Implementation; Suggestions-Documentation-on-Website; Suggestions-Tools;</p>
<p>Community's expectations of the potential contributors</p>	<p>Choosing-Documentation-Tools-by-Users; Conflict-Documentation-Style-Convention; Documenting-Formulae; Doc-Crowd-Source-Implementation-Issues; Issues-w-Tools-Documentation;</p>
<p>Standards and recommendations for crowdsourcing documentation</p>	<p>Accessible-SciPy-Project(ASP) Documentation; Docbooks; Conflict-Documentation-Style-Convention; Documentation-Formats-Tools-Poll; Suggestion-Documentation-Example; Suggestion-Standard-Importing; Developer-Doc-Implementation-Suggestion; Suggestion-Docstrings; Suggestion-Documentation-Contents; Suggestion-Modules-Documentation; Suggestions-Documentation; Suggestions-Tutorial-Instructions; Technical-Writer-Doc-Implementaion-Suggestion; Types-of-Documentation;</p>

Engaging the community	Accessible-SciPy-Project(ASP); Documentation-Engaging-Users; Implementation (Module-Submission-Review-Process); Implementation-Documentation-Website; Issues-w-Tools-Documentation; Motivating-Community; Project-Organisation-Suggestions; Spontaneously-User-Contributed-Documentation; Suggestion-New-User-Engagement; User-Suggestion-to-Contribute-to-Doc; Users-Contributions-to-Documentation; Ways-to-Engaging-Community;
Managing the documentation crowd-sourcing process	Accessible-SciPy-Project(ASP); Documentation-Formats-Tools-Poll; Implementation (Module-Submission-Review-Process); Implementation-Documentation-Website; Motivating-Community; Suggestion-Process-Contributing-Code; Suggestions-Organising-Process;

Table 7.2: Codes forming themes in Study 3 - SciPy / NumPy study

The additional data sources (the progress reports, the technical overview, the logs from the documentation server) were analysed separately. The progress reports (Harrington, 2008; Harrington and Goldsmith, 2009) and the technical overview (Van der Walt, 2008) were analysed using the codes developed during the thematic analysis of the mailing lists and interview transcripts. The analysis of the logs from the documentation server applied simple descriptive statistics, using Python scripts and the plotting library Matplotlib (Hunter, 2007).

7.4 Findings

The matter of improving documentation appears to be one of the very early considerations of the SciPy community. The archives of the mailing list are available from early 2002 onward. And discussions about documentation are visible as early as 2002. The interviews confirm this; although the study participants could not recall exact dates, they stated that documentation was considered as far back as they could remember. The rich discussion about documentation covered different types of documents,

from manuals and tutorials to docstrings embedded in the NumPy source code. The crowdsourcing effort was focused largely on the docstrings, and so the focus of this chapter is on them.

7.4.1 Reasons for crowdsourcing documentation

This section considers the reasons which led to documentation crowdsourcing. These findings come mainly from the analysis of the mailing lists archives, confirmed by the interviews. The interviewees stated that the documentation was incomplete and that the funding secured by one of the community members enabled the launch of the SciPy and NumPy Documentation Project. None was able to pinpoint who exactly came up with the idea of crowdsourcing documentation or when it occurred. The process of launching of the NumPy and SciPy Documentation Project was gradual; there were many events and discussions which contributed to setting up and organising the project. This section describes this complex process and, where possible, the reasons behind the events.

Almost from the beginning, documentation was lagging behind the development of NumPy. The source code was developing fast, and the documentation was not keeping up with it. Developers were not very keen on documenting the software. They were more interested in developing source code, and, since they were doing this in their spare time, they simply did not have capacity to write documentation. Developers recognized and appreciated the value of good documentation. However, their reluctance to work on documentation meant that the problem of incomplete or missing documentation was worsening.

Complaints about the documentation came mainly from new users. But experienced users also noted the issues with documentation when they had to introduce new users to the NumPy and SciPy libraries (for example, professors teaching students). Occasionally even experienced users struggled to find the desired functions, understand how they work and ensure that they really provided the required functionality.

An experienced SciPy user sent to the mailing list the results of the survey he ran among the students he taught using Python and SciPy. The major issue that the students complained about was the documentation. They wanted: “- *more extensive documentation*; - *examples for the routines*; - *help browser a la Mathematica*” [mailing list, first half of 2003]

The first ideas about fixing the issues with documentation by employing the community to produce documentation were noticeable on the mailing list in early 2003. Setting up a wiki was initially suggested (compare: Quote 7-1). The idea of crowdsourcing documentation started gaining support among the mailing list subscribers.

If only each scipy-user submits his simple example to test his most often used routine + a few lines of documentation this would help enormously! Improving

documentation (apart from testing and bug reports seems to me the place where “we” users could really help. [mailing list, first half of 2003]

A professional technical writer who subscribed to the NumPy mailing list applauded the idea of community-produced documentation. She strongly encouraged community members to join the effort and contribute to the project. Even though she was not a SciPy or NumPy developer, she had a lot of input into the discussions about the documentation.

Discussions about improvements were related to various types of documentation, from tutorials to the reference manuals. Docstrings were a type of documentation which was considered separately and could potentially benefit from the crowd-sourcing approach. Two things needed to be addressed, though: the stylistic guidelines for writing docstrings needed to be laid out clearly, and a technical infrastructure for collaborative editing was essential. Because docstrings were actually a part of the source code, getting the users to write them was a challenge.

It might be unavoidable that at the end some knowledgeable person has to go over suggested additions and incorporate them on a case-by-case basis? [mailing list, second half of 2005]

This technical difficulty (the solution to which is discussed later in this chapter) appeared to be something worth overcoming. It was suggested that many users might already have information which could be included in the docstrings (compare: Quote 7-2).

In the meantime, “Making Python Attractive to General Scientists” (later known as the Accessible SciPy Project or ASP) was initiated, and a Birds of Feather (BoF) session was held at the SciPy Conference in 2004. As one of the scientists declared:

Our short-term goal is to make Python attractive enough for ordinary, non-developer scientists to try it out, particularly those on platforms not well-supported by the current distribution systems. The long-term goals are to extend the community, and to draw from the extended community more and better documentation and application software. [mailing list, second half of 2004]

The BoF session helped to identify four areas requiring action: software issues, packaging (bundling together relevant software modules with the necessary dependencies), documentation and website. The requirements for the documentation were articulated:

It is important for docs to be well-written: clear, concise, grammatical, current, complete, correct, and visible. (...) Thus, it makes sense to coordinate this effort closely between the writers, relevant developers, and those planning the overall effort. The approach of writing shorter docs will enable participation by a larger group of writers (...) Of course, it will be important to provide good cross-references to other documents and the web site, and to keep these up to date. (...) [mailing list, second half of 2004]

As described in the introduction, one of the Accessible SciPy Project initiators, a university professor, secured funding to employ one of the main NumPy developers full time to improve docstrings and to coordinate the documentation crowdsourcing process. He did so because he saw how his students struggled with NumPy during one of his courses, because they were unable to find the right functions and to understand how to use them. He wanted to continue using NumPy for his classes but realized that it was impossible without improved docstrings.

He also announced on the mailing list that he was looking for someone competent with NumPy and SciPy to become a full-time docstring editor (or for two part-time editors). The professor explicitly asked the community for engagement and help in planning and organising the whole process. He was not a developer and hence was not sure how the docstrings could be edited externally. He asked for feedback on the ideas he put together and published on the SciPy website.

Even though the original motivation was to improve documentation for the students who struggled during their course, it was recognized that the initiative was useful for a wider community than just this group of students.(compare: Quote 7-3). The notion of expanding the community by improving documentation was emphasized by the professor who co-initiated the Accessible SciPy Project. The key was to provide enough information for users completely new to SciPy, so that they could easily get up to speed with the modules (compare: Quote 7-4).

In summary, there were several reasons which led to documentation crowdsourcing for NumPy and SciPy. The developers were not keen to spend their time documenting the package they were developing. Even if some documentation was written, it was difficult to keep up with fast progress in the source code development. New users struggled with the software; the lack of documentation meant that the learning curve for the newcomers was getting steep. Some experienced users occasionally found the lack of documentation equally challenging. At the same time, there was evidence that the existing users were writing bits of documentation for their own use. Knowledge that the existing users had about the software seemed to be a good reason to involve them in documentation writing. The idea started by two community members to make the software accessible for new users and to promote it within scientific communities was based on the assumption that major improvements in documentation needed to be done. One of these individuals not only secured funding to employ a full time documentation writer but also promoted the idea of engaging the community in documentation production.

7.4.2 Infrastructure for docstring crowdsourcing

This section focuses on establishing the infrastructure to support documentation crowdsourcing. It looks at the tools which were discussed in the community, and at the two systems initially set up: the

wiki and the livedoc site. These two systems inspired the development of a tailor-made documentation system used for the collaborative docstring editing. The draft timeline below depicts how setting up the infrastructure developed in relation to other events.

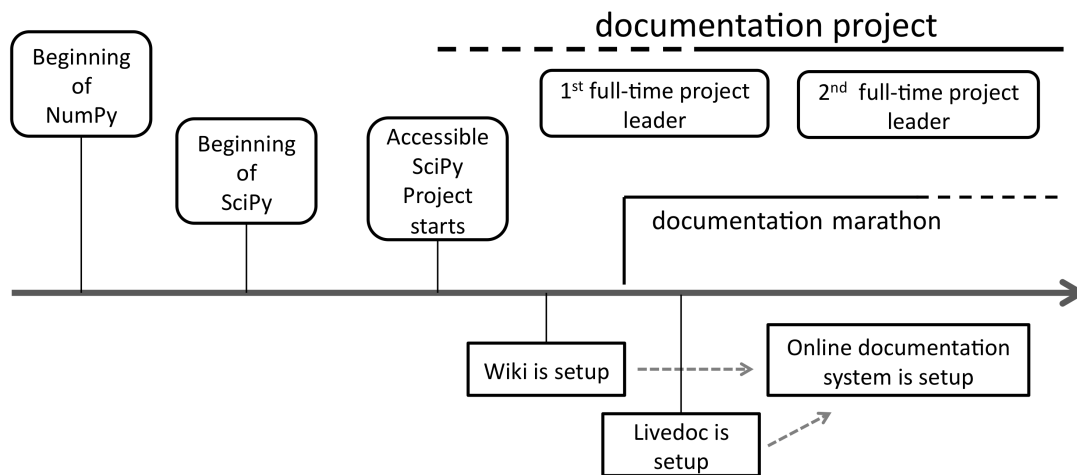


Figure 7.2: Timeline of the development of the documentation infrastructure.

Setting up the infrastructure to support crowdsourcing documentation was a complex process. The discussions on the mailing lists began with establishing which tool would be most suitable for working with the documentation. The choice of the tool for editing the documentation was one of the threads on the mailing list. The voices on the mailing list were divided between those supporting flexible and powerful tools which required some specialist knowledge, and those supporting tools that provided fewer features but were more user-friendly. The pros and cons of different tools were discussed both in the context of their appropriateness for editing documentation and in terms of the skills expected of the contributors. There was no single tool for editing documentation with which all potential contributors would be familiar. Hence, the argument that a particular tool would be ideal because it would be known to everyone, would not hold.

Several aspects of documentation production needed to be considered when choosing the documentation tool. The professional technical writer highlighted two of them: the ease of use and ability to include the documents in the download packages.

There is also a trade-off in ease-of-use vs. power and flexibility. To keep the barrier to entry low, it might make sense to use a simple tool (...) for short documents and a powerful tool (...) for longer documents. [mailing list, second half of 2004]

Apart from ease of use and compatibility with other tools and workflows, the tools for editing documentation had to accommodate writing and rendering mathematical formulae, in order to capture the science underpinning the software within the documentation. This would typically mean including a significant amount of formulae in the documentation. Some tools, such as Open Office, were thus completely unacceptable for use.

However, one of the main developers argued that equations will, in fact, take only a fraction of the documentation. He supported his argument with the observation that, in the Matlab documentation,

considered almost exemplary by many in the SciPy community, many documents did not contain equations. It should be noted that NumPy and other packages under the SciPy umbrella were perceived by the community as having great potential to be a free alternative to Matlab.

Another part of the documentation infrastructure to be developed was a wiki. Suggestions about putting up a wiki in order to enable the users to write documentation led to discussions about selecting the best type of wiki to set up. A variety of wiki-like systems were considered in order to find one which would fit the demands of the project.

The members of the community were encouraged to register with the wiki and work on documentation. It turned out that some were reluctant to do so as they felt they had no ownership over the wiki. They received an ensuring and encouraging response on the mailing list (compare: Quote 7-5). The wiki was set up not only as a documentation resource but also as a means to engage and expand the community.

The goal of the signups page is to make it easy for people to find each other: for lead authors to find people who will help them, for the community to identify who is taking part in what efforts, for low-level-of-effort volunteers to become hooked up with bigger projects, etc. [mailing list, second half of 2005]

The wiki solved the problem of collaborative editing. However, it still did not allow the volunteers to access the docstrings, and the changes made by the volunteers in the wiki were not reflected in the docstrings. The solution was to:

Set up a site (...) which has all the docstrings from scipy available in a hierarchical form.

** On each page there is documentation for a single function or class.*

** The documentation is separated into three parts:*

a) the one-liner

b) the docstring to be included in the scipy code

c) extra examples, documentation that will not be included in the code, but stay on the website.

** Every docstring in scipy contains a link back to the appropriate livedoc page so that users can edit it and/or find out more about the function. [mailing list, second half of 2004]*

Soon the first 'livedoc' page was constructed by one of the main NumPy developers; a documentation tree was built dynamically from the module docstrings. The docstrings were visible on the livedoc page. If the docstrings were changed, the livedoc page could be updated automatically. The idea was to make the hierarchy of the modules be reflected in the livedoc site.

This first livedoc site was met with interest. The community members involved in the documentation project started commenting and discussing various technical and design solutions. They gave their feedback about the livedoc site. As they gradually tried the site out, they were able to share their user

experiences. One of the main developers suggested that if the livedoc site allowed easy editing, it would then be an ideal tool to be given into the hands of the community, making it easy for almost anyone to get involved.

The livedoc site showed all docstrings from NumPy. Its drawback was that it was read-only. It allowed the community members to browse the docstrings, but not to edit them. The wiki did allow for editing but the edited changes were not automatically connected to the docstrings. Also, the wiki required the community members to create all the contents from scratch; the existing docstrings were not automatically fed into the wiki. The idea of combining the wiki and the livedoc site seemed to be an ideal solution. The idea, explained briefly in the email, was simple yet could potentially be very powerful enabling numerous contributors to be engaged in editing docstrings. Another developer also emphasized that this kind of infrastructure would significantly lower the entry barrier for those who would like to become more engaged with the project (compare: Quote 7-6).

Occasionally there was a message from a user who already has written some docstrings (for his own use) and was happy to contribute his work. This showed that infrastructure enabling docstring editing was essential (compare: Quote 7-7).

During the interview the developer who was hired as a main documentation editor and a coordinator of the community work explained that maintaining the effort of the community needed the support of relevant tools. Additionally, developing the infrastructure was an interesting change from writing documentation 24/7.

...the first goal was to add documentation but we realized that it doesn't help just to add the documentation. You need to have a sustainable effort and that's why we added all these tools instead. [Interviewee B]

In summary, the technical infrastructure to support the documentation crowdsourcing process evolved over time. A variety of tools were discussed together with their properties. The choice of the tools was a trade-off: powerful tools could be a barrier due to their steep learning curve, and tools that were easier to use potentially did not allow the user to deal with all tasks related to documentation. Setting up the wiki to accommodate the collaborative work turned out to be only a partial solution. It was not connected to the docstrings and hence did not allow the volunteers to edit them. Another system set up in the meantime, the livedoc site, showed the docstrings in a hierarchical tree and could be updated with any changes in the docstrings automatically. However, it was limited to read-only access. Finally, an idea emerged to develop a system which combined the two properties: that of a wiki enabling the community to work collaboratively, and that of the livedoc site connecting directly to the docstrings in the source code.

7.4.3 Community's expectations of the potential contributors

This section considers community members expectations about the knowledge and skills of the potential contributors to the Documentation Project. Some discussions about the expectations of the contributors started in the context of discussing the editing tools and documentation formats. The underpinning theme in these discussions was related assumptions by different members of the community about the contributors.

Because there was no tool which would be known to all potential contributors, the choice of a particular tool would result in a temporary advantage for one section of the community. Potential contributors who were not familiar with the tool would not edit documentation until they gained the relevant skills and were able to use the tool. However, this upfront exclusion of some of the contributors was perceived by one of the developers as a desirable effect rather than a problem.

There was a discussion about using LaTeX as a tool for editing documentation. This discussion was not only about choosing the right kind of tool. It also revealed expectations about the potential contributors.

As I have understood from earlier discussions that LaTeX may scare people away from writing documentation for Scipy. But let me ask may be a bit harsh question: are these people able to produce good quality technical documentation for Scipy if they reject to learn LaTeX basics? After all, LaTeX is not that difficult to use, especially when we can provide documentation templates to get started easy, and there are nice LaTeX frontends available for Windows as well. [mailing list, second half of 2004]

This statement was met with a strong rebuttal.

Harsh answer: Yes. We are talking about busy people with day jobs, for whom SciPy is only a means to an end. Their knowledge of SciPy and their ability to write good quality documentation in no way depends on their ability or willingness or time available to learn or use any other specific tool. [mailing list, second half of 2004]

The technical writer argued that the documentation tool should be transparent. That is, that the contributors should be welcomed to use any tool they felt comfortable with.

Content is king. While standardizing on a tool or toolset is important, nobody should let that stand in the way of generating content. [mailing list, second half of 2004]

The idea that using a tool for editing documentation might serve as a filter to select documentation contributors with a desired level of skills and knowledge – and would possibly lead to excluding those who could help address the documentation problems – raised concerns, because the the project needed many contributors.

Eventually the developer who initially asked if the contributors who do not know LaTeX should be considered for documentation became convinced by the arguments provided by others.

I agree with the general pragmatic viewpoint that docs should be accepted regardless of the format and LaTeX should not be a filter. [mailing list, second half of 2004]

There was an expectation that the experience of contributors who wrote about science on daily basis would allow them to engage smoothly in documentation crowdsourcing, because writing documentation would be familiar ground for them. However, one of the interviewees disagreed; he pointed out that a different set of skills was needed to write documentation than to write research publications or teaching materials. Another interviewee remarked that, since the users knew and used the software, this was already a good selection criterion for documentation writing.

In summary, the discussion on the mailing list revealed that there were different expectations about the skills that the potential contributors should have. One stance expected contributors to be fluent in using tools such as LaTeX, because those who struggle with tools like LaTeX should not be trusted with documentation writing. However, this opinion was met by several contrasting opinions. The argument was made that screening the potential contributors based on their knowledge of advanced editing tools was not the goal of the crowdsourcing process. There was also an expectation that the scientific and academic background would help the contributors in producing documentation. On the other hand, documentation writing required slightly different skills than writing research papers or teaching materials.

7.4.4 Standards and recommendations for crowdsourcing documentation

This section considers how the members of the community set up the standards and recommendations for the documentation which was to be crowdsourced. The discussions about standards and recommendations included the stylistic guidelines as well as the contents of the documentation. These discussions took place mainly on the mailing lists. The matter of standards and recommendations was also raised by the Accessible SciPy Project. After the Birds of a Feather session at the SciPy Conference in 2004, the scientists leading ASP declared explicitly that standards for documentation were one of the goals for ASP.

A professional technical writer suggested dividing the documentation into smaller documents in order to support good organisation: A short document has less opportunity to be poorly structured than a long one. A collection of short documents can be assembled into a helpful organisation as it evolves. [mailing list, second half of 2004]. As soon as the initial wiki was set up, the matter of standards and style was discussed, and it was agreed that the wiki should include stylistic guidelines to which the contributors could refer at any time.

A proposed standard for docstrings was based on the documentation style that one of the developers had used over years. The standard covered things such as: inputs, outputs, algorithm, notes, authors, examples, comments, tables, footnotes and so on. The suggested standard was met with support and appreciation on the mailing list. The subsequent remarks and suggestions for small changes were not any kind of strong criticism. For example, one of the developers suggested that storing the authors names was a redundant piece of information as version control already recorded the authorship. The standard was described in a plain text file HOW_TO_DOCUMENT.txt which was publicly available online.

Nevertheless, the discussions on the mailing list about the standards, formats, guidelines and recommendations show that these matters were not straightforward to establish. There were cases in which, after a democratic exchange of opinions and views, an executive decision had to be made. For example, roughly a month and a half after the docstring documentation project was launched, the developer coordinating the contributors' work sent an email to the list which included a number of suggestions for changes both for the infrastructure and for the proposed docstring standard. All these ideas and suggestions were based on his experience of running the project for these few weeks. The scientist who was heading the project (and who secured the funding) suggested that the decision should be made based on an online poll.

However, the discussions continued even after the poll, and eventually it was suggested that two key people should make a decision: the developer employed to write the documentation, and the developer who built the infrastructure for the docstring documentation (compare: Quote 7-8). The first of the two developers addressed replied:

I have updated the documentation standard to reflect the way we'll be using the shorthand forms from now on. (...) Sorry for annoying you, but it's over now. We've sorted out the problem that kept us from editing for the past two days: all systems go![mailing list, first half of 2008]

In summary, agreeing on standards for the docstrings involved several discussions on the mailing list. Standards were necessary as the contributors needed to know how to write documentation. The standards were related to both the format of the docstrings and their contents. Decisions about the standards were discussed following the "democratic" spirit. However, the final decision was left to two people: the full-time documentation editor and the developer who set up the documentation system.

7.4.5 Engaging the community

This section considers how the members of the community were engaged with the documentation crowdsourcing process. Ways to convince the contributors to join the initiative are discussed, along with approaches to keeping the volunteers motivated.

Community engagement was highly valued and encouraged. Both the developers and the user who secured the funding for the documentation editor thought that the community's engagement was important. Everyone's efforts, including those of new NumPy and SciPy users, were welcomed and supported publicly.

Being a 'newbie' is maybe the best time to write documentation – while you still know what new users need to know, and how to explain it to them simply. [mailing list, first half of 2007]

Community members were not only encouraged to produce documentation but were also asked to comment on ideas related to organising the process, documentation standards and other recommendations (compare: Quote 7-9). It was emphasised that in collaborative and community-driven work it is essential to take on board comments and suggestions from one's peers.

The infrastructure developed for editing the docstrings provided a level of security for the original source code but at the same time allowed many contributors to become involved. In order to be able to edit and write docstrings in the system, the contributors had to register and then send an email to the mailing list requesting full access. This additional step potentially made the registration procedure too long for some contributors.

If you are an ordinary user and let's say that you want to just to correct some typo, then you might not do this because the registration has these steps. (...) I think the only evidence that this has happened is that there have been slightly more people creating the accounts so solving this first step of the registration than those who have sent mails to the mailing list. [Interviewee C]

The infrastructure itself along with the community willingness to contribute did not guarantee that documentation would be produced in the long term. The presence of a project leader who coordinated the effort was essential to ensure the sustainability of the project (compare: Quote 7-10).

In order to keep up the momentum for the documentation writing, the Documentation Marathon was introduced. The Marathon was modeled on coding sprints which were very popular within the community. The Documentation Marathon was organised to bring together documentation contributors which resulted in their increased productivity.

Four months into the documentation project, the scientist who headed it presented a summary of progress. He described what was ready in terms of the infrastructure and the organisation.

there are 2323 NumPy objects with docstrings. Of these, 1464 we deemed "unimportant" (for now). These are generally items not seen by regular users. This left 859 objects to document in this first pass. We've done 24% of them at this writing. [mailing list, second half of 2008]

At the end of 2008, so about six months after the Documentation Project was launched, its main initiator summarized the results of the work so far. He provided a detailed breakdown of the different statuses of the edited docstrings along with a general summary. He specifically indicated which modules still needed to be documented and strongly encouraged the user community to join in and edit them. Again, the emphasis was made that the contributors did not have to be the developers of NumPy and SciPy.

The statistics and public recognition regularly published on the mailing list or at the SciPy Conference were not the only tokens of appreciation.

We promised a cool T-shirt to everyone who has written 1000 words on the doc wiki. [mailing list, second half of 2008]

In mid-2008 it was reported that 16 people received the T-shirts. It showed that in crowdsourcing the distribution of the work is not even. Yet this was no revelation to the Documentation Project organisers.

There were people that would write maybe a few pages of docs that they were interested in and then didn't write any more. Like most projects 80% of the work was done by 10% of the people. Most people just cranked down the documents. [Interviewee D]

The analysis of logs from the Documentation Project server confirmed this observation. The figure below shows the percentage of contributors grouped by the number of words they edited in the documentation system.

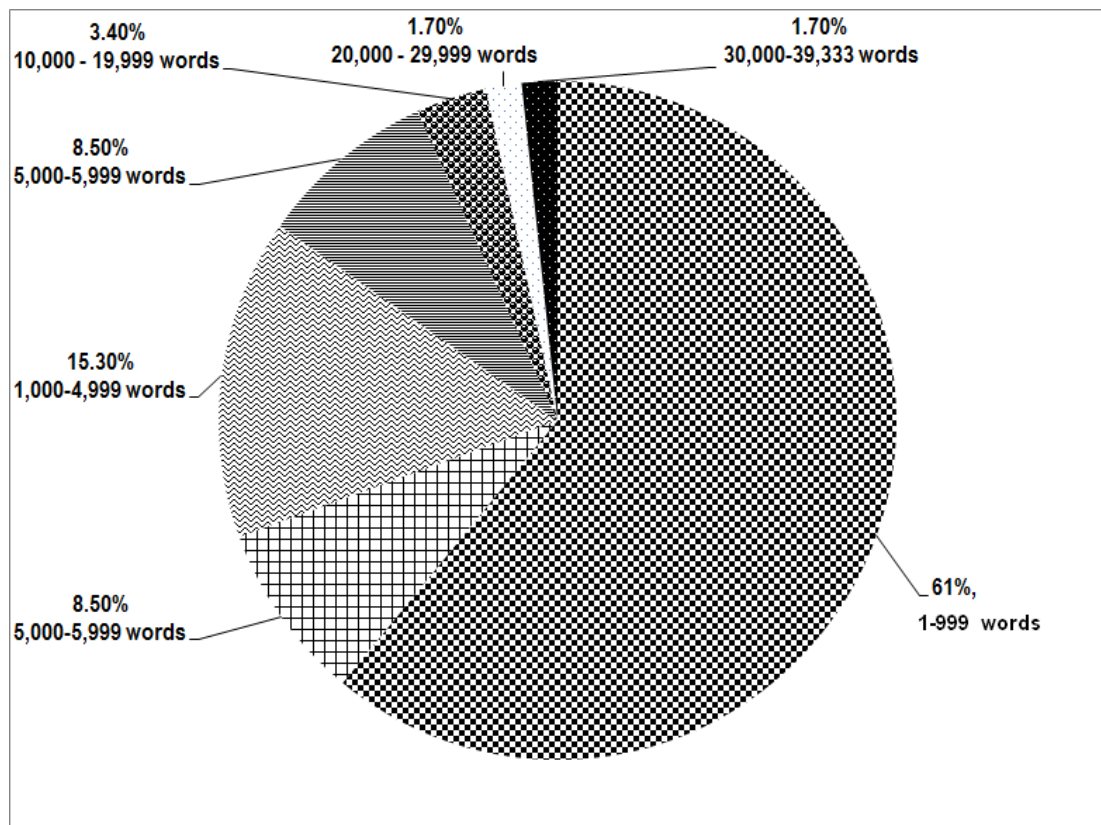


Figure 7.3: The number of words edited in the SciPy and NumPy Documentation Project by the contributors.

The Documentation Project organisers did not expect initially that the community would engage eagerly and in large numbers.

I think the threshold was about 30 people and I told them that I don't think that we are going to have that many signups for the first 2 months. I think the bet was on ice-cream. I lost that bet. Lots of people signed up. [Interviewee B]

I personally did not have a view that we would get 75 people involved in it. That number shocked me. [Interviewee D]

The logs from the Documentation Project server show that until mid-2012 there were about 250 registered contributors accounts (as shown on the figure below).

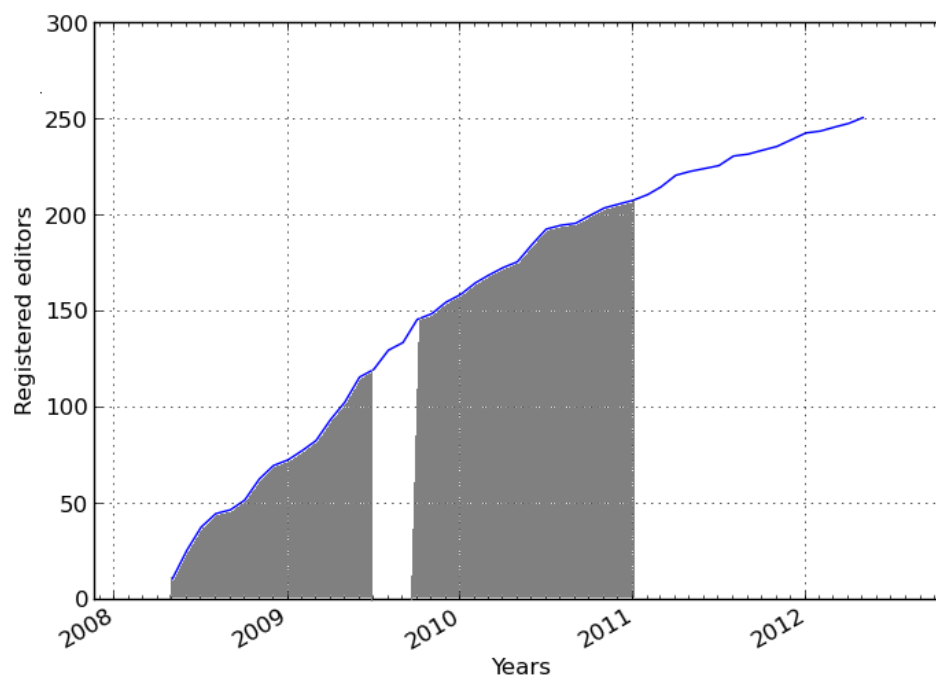


Figure 7.4: The accumulative number of registered NumPy/SciPy Documentation Project contributors. The grey area indicates the periods of time when the full-time editor was employed.

In the interviews, the Documentation Project was perceived as a successful way of expanding the SciPy and the NumPy contributors community. Providing new ways in which the members of the community could contribute meant that those who were unable to join earlier found their niche.

It was a particularly useful way of engaging those who wanted to help but felt that they were unable to do so because, for example, they did not know how to contribute to the source code.

I'd say it really lowers the energy value for contributing to open source software. I think it's a very important point. [Interviewee F]

The Documentation Project proved to be a low entry barrier for all those who were eager to help but did not know how to start (compare: Quote 7-11). The core developers were aware of that, for many users, contributing to the source code was a high entry point. At the same time, they recognized the value of

the skills which the users had that would be very useful in writing documentation (compare: compare: Quote 7-12) – or in other contributions to the community.

One message that I often have in the conferences is that writing code is not the only way of contributing to the SciPy and NumPy community. There are other ways like writing documentation, teaching, organising conferences and it can actually help you to gain better technical skills as well. [Interviewee F]

In summary, community engagement involved two aspects. The first was providing an infrastructure which was easy for the contributors to use. Lowering the entry barrier helped to encourage more users to join the project and write documentation. The second aspect was related to social interactions. There was a welcoming approach towards anyone who wanted to contribute and write documentation. In the long term, community engagement became one of the tasks of the full-time documentation editor. To keep the community engaged and motivated, up-to-date statistics on the progress with the documentation writing were provided. Public recognition was given to those who worked on documentation. Crowdsourcing documentation showed the SciPy and NumPy community that there were multiple ways of contributing to the software, not only limited to developing source code.

7.4.6 Managing the documentation crowdsourcing process

This section considers how the NumPy and SciPy Documentation Project was managed. Managing documentation crowdsourcing was a complex task. One of the key things was to lay out clearly to the community how they could help. The step-by-step instructions were provided as soon as the Documentation Project started. The clear plan of what to do was especially important for new contributors who wanted to help with documentation.

- 1. Write some docstrings on the wiki! (...) you must know NumPy, the function group, and the function you are writing well. You should be familiar with the concept of a reference page and write in that concise style. (...) See the instructions on the wiki for guidelines and format.*
 - 2. Review others' docstrings and leave comments on their wiki pages.*
 - 3. Proofread docstrings. Make sure they are correct, complete, and concise. Fix grammar.*
 - 4. Write examples ("doctests"). Even if you are not a top-notch English writer, you can help by producing a code snippet of a few lines that demonstrates a function. (...)*
 - 5. Write a new help function that optionally produces ASCII or points the user's PDF or HTML reader to the right page (either local or global).*
 - 6. If you are in a position to hire someone, such as a knowledgeable student or short-term consultant, hire them to work on the tasks above for the summer. We can provide supervision to them or guidance to you if you like.*
- [mailing list, second half of 2008]

The workflow for documentation production was one issue. The contributors needed to know what had already been edited and was ready to be checked in. They wanted to know if any of their edits were

not accepted and needed further editing. The workflow was designed and integrated with the docstring editing infrastructure. The docstring was first qualified as “needs editing” (which could mean both editing and writing from scratch), then as “being written” and then “needs review”. Then it could either be qualified as “reviewed but needs work” or “reviewed needs proofing”. Finally, it was qualified as “proofed” and ready to be applied in a patch.

However, it turned out that it would be very difficult to fully exploit the proposed workflow. In particular, reviewing of the edited docstrings was difficult to achieve as the reviewers were usually the core developers who had other priorities. But again the fact that the reviewing part of the workflow did not work that well was not perceived as causing harm. In general the reviewing task itself was challenging and required different skills from the reviewers.

I have come to recognize two forms of review: content and presentation. (...) we should consider requiring both kinds of review. It will slow us but will improve the product a lot. Some of our experts are not great writers and some of our great writers are not experts.[Interviewee C]

Another part of the workflow which proved to be difficult to implement was proofing. Due to the lack of time, the proofers were not able to check all docstrings.

Registered contributors could comment under each of the edited docstrings and openly discuss proposed changes. It was a solution similar to the one used in issue trackers. The figure below is a screenshot from the documentation system showing an excerpt from a discussion between the contributors under one of the edited docstrings.



Figure 7.5: An example of a discussion under an edited docstring on the documentation project website. The names of the contributors are blanked off to ensure anonymity.

As the work was gaining momentum, and more and more volunteers were joining the effort, management and organisation became crucial. The amount of docstrings to edit was large and, apart from the NumPy docstrings, SciPy docstrings were also available for documentation. The professor who secured the funding for the Documentation Project suggested that:

Guide anyone who is not a maintainer of SciPy code to contribute effort to the NumPy docstrings until they are done, then the SciPy docstrings. [mailing list, second half of 2008]

After the contract for the first full-time documentation editor ceased, the professor who originally hired him secured more funding. He put out a call for a second full-time docstring editor (compare: [quote](#)). This time the call clearly specified that a considerable part of the job would include coordinating the community's effort. The newly hired documentation editor and project coordinator soon announced the work plan for the coming months. Based on the experience of the previous six months, he proposed that the volunteers should be organised in teams. However, those who preferred to work individually could still do it.

Securing the funding for employing a full-time editor and a project leader appeared to be one of the key factors which helped in organising the Documentation Project and making it successful.

this would not have happen without funding. Nobody in the scientific community was willing to really (...) to go deep and flesh out all the details about the documentation. (...) there are probably multiple things that are just a barrier to entry where nobody would pick up on things. [Interviewee A]

The importance of having a project leader, a person who would be a full-time coordinator, was emphasised by four other interviewees. The coordinator was particularly important at the beginning of the project. This person had to put a lot of effort into engaging the community. The end result of having a number of volunteers actively contributing to the project was thanks to the intensive efforts by the main coordinator and the rest of the core team who set up the project.

In summary, managing the collaborative effort of producing documentation was almost a full-time job. There were a number of tasks which the community members were encouraged to fulfill. The process of producing docstrings was divided into several activities, from editing them, to reviewing and proofing. These different activities were facilitated by the technical infrastructure. When the second full-time documentation coordinator was sought, the job advert explicitly specified that his role would involve a considerable amount of management effort. Securing funding for employing a full-time editor and a project leader was a key factor for success, one noted by all interviewees.

7.4.7 Motivations for joining the documentation crowdsourcing project

The previous sections presented reasons for launching the NumPy/SciPy Documentation Project and mentioned the benefits which it brought to the community. This section considers a more *individual* perspective on motivations for helping in the Project.

One of the volunteers simply liked documenting. He explicitly said that he did not have skills or time to develop the source code. However, he was eager to produce documentation. He not only liked doing that but already had some experience.

Another user clearly saw that producing documentation could bring him many benefits:

*I'm not actually *that* experienced a user (...) But I can learn enough to write documentation, and it is forcing me to improve my NumPy skills and knowledge (...). My point is this (...) you just have to be willing to expend some effort learning enough to write a document. And that effort will pay dividends - the best way to learn something is to teach or document it. [mailing list, second half of 2008]*

It was also clear that in an open source project if an individual wanted to have anything changed, it could only be done by taking some action oneself. The lack of documentation made using SciPy a frustrating experience. Contributing to the Documentation Project was a great opportunity to deal with this frustration, and to replace it with the reward of seeing the results of one's work.

As one of the interviewees explained, motivation also came from the feeling that the users wanted to give something in return for being able to use the software completely for free.

There is a human dynamic in that which is very appealing. You want to get engaged. Obviously there are people who simply want to get things for themselves but most decent human beings when they are treated with generosity, there is a natural response to at least return a little bit. But the problem is, if in order to return in kind, you have to cross very high barriers, you may end up giving up. I think the point of creating a system like this was '(...) Let's lower the unnecessary barriers so that they can easily contribute back and so that we can take the contributions and integrate them. [Interviewee G]

The developers' open and welcoming approach to users, even inexperienced ones, was also considered to be a motivational factor. The fact that the developers were willing to interact with the user community and provide opportunities for the users to help was a big incentive for users (evident in both the mailing lists and interviews).

However, involving the community in the Documentation Project still posed challenges.

I think it takes effort in our community to convince people that it is important to document our tools(...) because that doesn't give anyone points up to tenure or up for promotion or in applying for a job because they cannot put that in their CV, that is hard sell in our community. [Interviewee G]

In summary, the personal motivation for contributing to the documentation crowdsourcing varied. Contribution provided opportunities for reflection, learning, and code improvement. Another motivation was the bothersome state of documentation and the realisation that, if someone want improvements in an open-source project, they must act themselves. The need to give something back to the community, to be useful and contribute, was also motivating. On the other hand, documentation production was not something which could advance one's research career. Therefore, it was likely that not all members of the community had the same level of motivation to contribute to documentation production.

7.5 Discussion

The findings presented in the previous section covered a number of themes. The discussion relates to all of them but is mainly guided by the four main research questions underpinning this empirical study:

- 1) *What may lead to crowdsourcing scientific software documentation?*
- 2) *How can crowdsourcing scientific software documentation be organised?*
- 3) *What are the benefits of scientific software documentation crowdsourcing?*
- 4) *What are the challenges of scientific software documentation crowdsourcing?*

Section 7.5.1 discusses the reasons and motivations which may lead to documentation crowdsourcing. The next section 7.5.2 looks at the organisation and management of the documentation crowdsourcing process. Section 7.5.3 focuses on the benefits which may emerge from engaging the community in documentation production. The following section 7.5.4 discusses the challenges which scientific software documentation crowdsourcing may bring. Finally, the discussion considers whether the crowdsourcing process improves documentation.

7.5.1 What may lead to scientific software documentation crowdsourcing?

The first thing that comes to mind as the answer to the above question is: insufficient documentation. But it is worth looking beyond this answer and understanding why the documentation is insufficient. Primarily it was the lack of time of the developers and their general reluctance to document. These reasons have been noted in previous studies presented in this thesis. They are also present in the literature (for example, limited time and required effort are the main reason against documentation production according to Nguyen-Hoan et al. 2010). Additionally, in the case of NumPy and SciPy the software was developing in a very fast pace and documentation was simply lagging behind. This situation was painful for the software users, in particular the newcomers. The developers not only appreciated this issue but also recognized that many users had skills which allowed them to produce high quality documentation. In addition, some of the users actively shared the documentation they wrote for their own purposes. The evidence from the mailing lists archives clearly shows that the developers were very appreciative of users' documentation. Even users who were not very active on the mailing list were

happy to share any documentation they wrote. At the same time they typically expressed their understanding of the fact that the developers did not have time to provide documentation. This atmosphere of mutual respect and appreciation of each others' efforts led to the ideas of crowdsourcing documentation.

The developers did not claim exclusive ownership of the software nor of any other related contributions (such as documentation writing). Overall, there was a strong sense of collaborative ownership, even if only a small group of developers had write access to the repository. The core developers nurtured collaborative ownership by encouraging the community members to express their opinions, give suggestions and comment on various aspects of SciPy. Potentially the wide community's engagement in documentation production could result in finding and exposing a number of bugs in the software. This could be a benefit for everyone as the software could be improved and made more robust.

The users who contributed to the documentation were also eager to help. Often only their (perceived) lack of skills was preventing them from other forms of contribution such as developing the source code. The eagerness with which users replied to the appeals to join the project indicated that they felt that they were a part of the community. This is similar to what Bagozzi and Dholakia (2006) found in their study of Linux User Groups; they noted that one's activities within the user group community were an integral part of maintaining one's social identity. The SciPy community members also felt that they wanted to give something in return for being able to use free and efficient software. This resonates with what Lakhani and von Hippel (2003) found; in their study of open source communities, one of the main reasons for helping other users was the need for reciprocation.

The SciPy and NumPy Documentation Project is an example of how one user's need (in this case, a professor's need to use the software in his teaching) can act as a catalyst for the community. As typically happens in open source projects (Raymond, 1999), "scratching one's personal itch" led to implementing a solution which benefited the larger community. Even though the ideas of documentation crowdsourcing were already present among the community and some initial infrastructure was established, it was this user's injection of funding (and hence a full-time documentation coordinator) that pushed things forward. In fact, his goal extended beyond just using SciPy and NumPy for his classes; he wanted to make the software easily accessible for new users and, in result, extend the user community.

All these events, approaches and attitudes were mutually reinforcing. The users were happy having useful software for free and felt gratified to share what they produced on their own. The developers not only appreciated these efforts but also noticed that crowdsourcing documentation could take a big burden off their shoulders. A few community members started building up initial infrastructures to accommodate users' work. And soon after, a dedicated user with secured funding stepped up. It could

appear as if this was lucky coincidence. However, the evidence found in the data shows that providing the right atmosphere and building the community can turn out to be extremely beneficial and create an environment in which documentation crowdsourcing may flourish.

7.5.2 Organisation and management of scientific software documentation crowdsourcing

This case study shows that organising documentation crowdsourcing into a smoothly running machine is a complex and difficult task. First, a technical infrastructure that would support the process had to be established. Second, the existing crowd knowledge and existing bits of documentation needed to be harnessed. The key was to give the contributors easy access to write and edit the documentation and at the same time to minimize the possibility of potential errors, mistakes and confusion. Wiki-like systems are probably the most suitable solution. As Dagenais and Robillard (2010) showed, wikis are quite often used in open source projects for documentation. But, in their study, Dagenais and Robillard (2010) looked mainly at the contributions made by the developers. There may be other infrastructures for crowd documentation management and organisation such as so called Question and Answer websites, for example Stack Overflow (www.stackoverflow.com, Parnin et al. 2005). However, these websites do not provide an opportunity to connect the documentation to the source code, and they don't enable any further manipulation of the documentation. The evidence from the mailing lists archives show how collaboratively the community members came up with the idea of connecting the functionality of the livedoc site with the docstrings with the wiki. One of the developers (with support from a few others already involved in setting up and maintaining the documentation wiki) stepped in and committed his time in implementing a tailor made solution. It should be noted that this particular individual did not receive any funding for this activity. The dynamic in this example and the professors funding (above) was similar: an individual took the ideas, suggestions, and comments on board and turned them into action.

The choice of tools and standards not only affected format and flexibility of documentation, but also had an impact on the selection of potential contributors. The discussion on the mailing list relating to the formats and editing tools was in a large part a discussion of the expectations and assumptions about contributors. All tools have the power to include or to exclude some of the volunteers. Again, this discussion, even though very intensive, showed that the SciPy community was very inclusive and did not try to position itself as a kind of elite group. For example, there was explicit recognition that someone may not know LaTeX but still be a very good scientist and potentially a desirable documentation contributor. The notion of a welcoming atmosphere, open even to inexperienced novices was valued.

Essentially anyone could edit the docstrings. There was no background check on anyone who registered

on the system and asked for permission to edit. The only reason for making people ask to be given the full editing credentials was to avoid spam. It not only shows that the overall atmosphere was very welcoming but also that there was a high level of trust in the community. Of course, what helped was the fact that the infrastructure actually had security gates - only administrators had credentials to create and submit patches. Nevertheless, running any kind of checks on people signing up was never a concern.

Establishing a workflow to help direct the effort of the volunteers turned out to be essential. It was not necessary at the very beginning when there was a relatively small number of existing docstrings and a large number of those that were supposed to be written. The workflow was designed very early and integrated into the infrastructure. Even though it was a relatively simple design, it did not work exactly in the way in which it was foreseen. The review and proofing loops were often skipped. This shows two things. First, that even the simplest workflow or any other management design may be difficult to apply, in particular in an open source project in which people essentially work in their free time. Second, that skipping these two steps was not damaging to the whole project. The Documentation Project organisers were understanding and appreciated the fact that the reviewers were unable to commit more time. But documentation was still produced, and docstring patches were still applied on the source code.

All of the data sources showed the high importance of the role of the project leader in organising and managing the documentation crowdsourcing. The project leader divided work, assigned tasks, negotiated and helped to resolve conflicts, and motivated the volunteers. In crowdsourcing the power is indeed within the crowd – a large number of people where everyone is keen to contribute even a little bit. It should be noted that, in the case of SciPy and NumPy, what helped to make the project successful was a small group of individuals who took responsibility, and in particular the project leader. In the case of the SciPy and NumPy documentation project, the project leader had to be an experienced developer who also had a good relationship with the user community, in order to be able to manage their work.

7.5.3 The benefits of scientific software documentation crowdsourcing

The biggest benefit of the documentation crowdsourcing was improved software documentation (discussed in a separate section, below). This section discusses other benefits for which evidence was visible in the data.

The standards and guidelines for the NumPy and SciPy documentation were a benefit, even though the documentation system was not one of the main goals for the Documentation Project. However, it turned out to be extremely useful in helping the community clean up the format and the contents of the documentation. It made it possible for novice volunteers to engage fairly quickly and write documentation, without having to ask the more experienced contributors to introduce them to writing documentation.

It reduced the problems with incompatible bits of documentation. Adhering to the guidelines and standards meant that those who were reading the documentation were likely to receive a coherent piece of information about the software. As mentioned by one of the interviewees, the standards also remained in use and were picked up by other scientific Python packages related to SciPy. It is possible that similar documentation formats and contents organisation, as well as harmonized terminology, will potentially make the documentation in these scientific software packages as well as SciPy and NumPy easier to manipulate and process in the future.

Another positive outcome of the documentation crowdsourcing process was that users who until then did not perceive themselves as being able to contribute to the project became more confident and became strongly engaged. In a few cases, they developed from users into user-developers and started writing source code. It can be said that, in effect, the community of user-developers expanded.

The community was probably inclusive and friendly before the Documentation Project started. But this inclusivity and friendliness was reinforced by the Documentation Project, which encouraged everyone to contribute, regardless of their previous experience in NumPy or SciPy. The efforts of those who got involved were publicly appreciated. The contributors could see the effects of their work in a very short time. Some discussions were intense, but there were no signs of impoliteness or attempts to undermine someone's opinion or competence. There were conflicts and disagreements (discussed further in the next section); however, the process that the SciPy community adopted in the Documentation Project appears to help building a friendly and inclusive group.

7.5.4 The challenges of scientific software documentation crowdsourcing

As discussed above, documentation crowdsourcing worked well when the project leader was present. The challenge for successful community engagement is finding the right person to manage and organise the group efforts. Such a person should have skills which combine both technical knowledge and the ability to collaborate with and motivate people. As this case study shows such people can be found within the community members. However, what is more challenging is whether these people can actually commit to such demanding work. In the Documentation Project both leaders were employed full-time and received remuneration. It is certainly not a typical case for open source projects which sometimes struggle even with, for example, paying for hosting services. The question is whether without these full-time leaders the SciPy and NumPy documentation crowdsourcing would achieve the stage it achieved. How could this work if there was no funding? Potentially, the tasks of the leader could be shared between two or more people who would do it in their free time. However, they would have to achieve a high level of coordination and collaboration between themselves before helping others to collaborate. It may be that solving the question of leadership is problematic for any crowdsourcing process.

The tasks the project leader had to face were challenging too. Keeping up the momentum was a hard and, essentially, full-time job. Initial enthusiasm and excitement can easily decline. When problems and challenges appear, the volunteers may lose their interest and withdraw. Maintaining momentum required keeping an eye on what was done and providing feedback. The feedback was positive, enthusiastic and expressing a lot of appreciation for the volunteers' effort. At the same time, the feedback had to be meaningful. It all required engagement from the project leader.

This case study showed user engagement and enthusiasm for producing documentation. People saw and believed that contributing to the project makes sense. However, as one of the interviewees pointed out, writing documentation is probably one of the least appealing tasks in open source projects. Although producing documentation provides opportunities for reflection, learning, identity-building in the community, and software improvement, producing documentation is not an activity which would make a particular improvement to an academic CV. The Documentation Project did a lot to make this activity more appealing and "sexy", as one of the interviewees put it. As previous studies showed, writing documentation aids reasoning about the software itself. It helps to clarify ideas and notice bugs. Documentation crowdsourcing may potentially help to improve software.

7.5.5 Does crowdsourcing improve documentation?

The evaluation of the effectiveness of scientific software documentation crowdsourcing was not one of the main goals of this study. However, the question "Does it actually improve documentation?" cannot be avoided. Although the case study data, in particular the interviews, contain some evaluation statements, it is nevertheless difficult to evaluate the SciPy and NumPy Documentation Project fully.

The question of what is good documentation has been around for decades (Rettig, 1991). Certainly in raw numbers, the outcomes of this documentation crowdsourcing process are impressive. From 8,658 words, before the project began, the NumPy reference pages increased to over 140,000 words (as of July 2012). All these words were edited and checked by skilled and knowledgeable members of the SciPy community. Most of them were users themselves and hence were able to understand other users' needs. It may be safe to assume that the new docstrings are informative and useful for other users.

It is possible that the improved documentation attracts more users. The NumPy package is now widely used. The figure below shows the gradual increase in the number of downloads of NumPy from SourceForge¹. The drop in 2013 is probably caused by a change in installation: the current recommendation is to install NumPy using either package managers or widely available packages such as Anaconda².

¹<http://sourceforge.net/projects/numpy/files/stats/timeline?dates=2008-01-01+to+2013-08-21>

²<https://store.continuum.io/cshop/anaconda/>



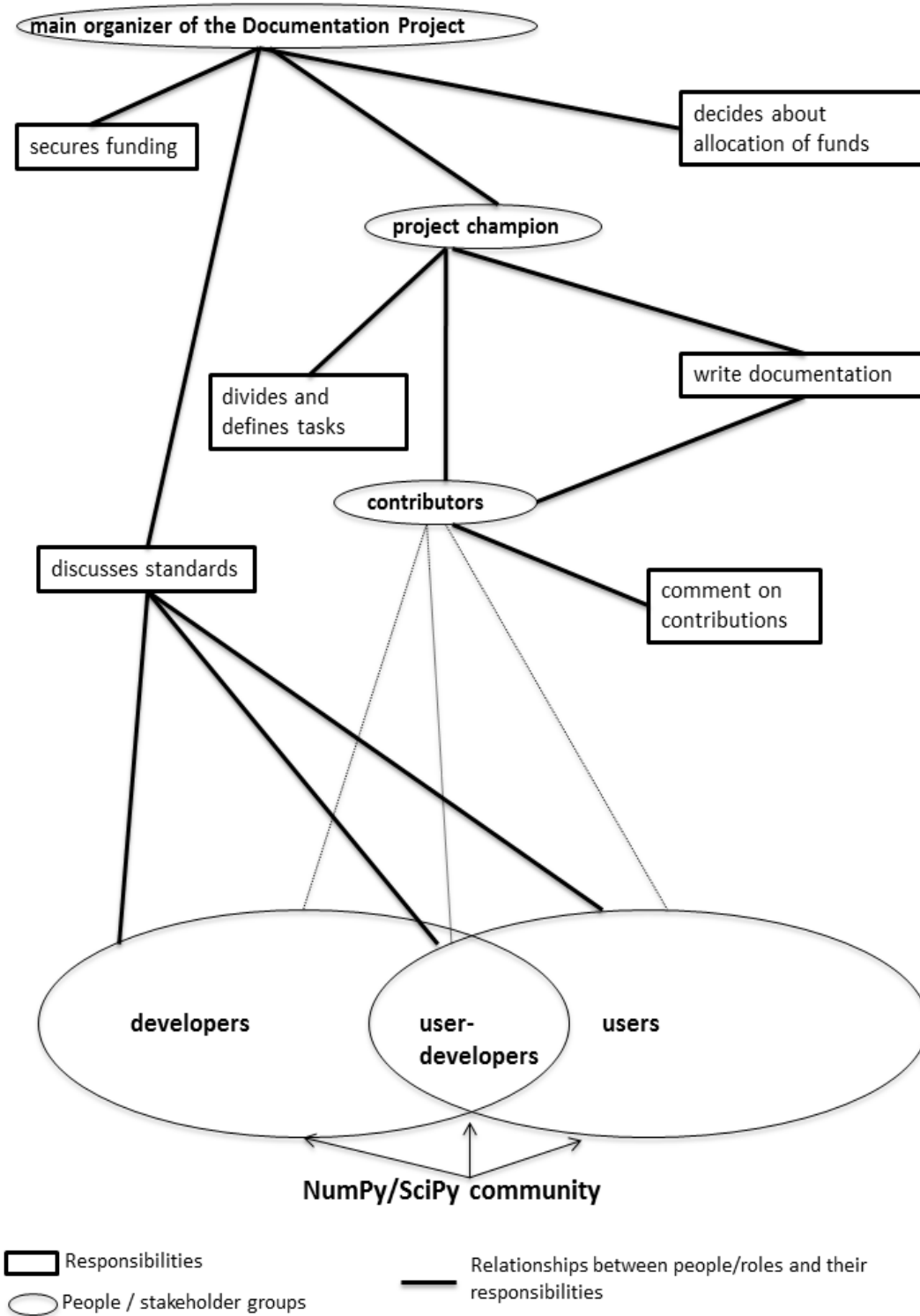
Figure 7.6: NumPy installation packages downloads from SourceForge.

However, it is difficult to attribute this success to the improvement of the documentation only. In the recent years Python itself gained massive attention among scientists in various disciplines. What may be said is that, without good documentation coverage, NumPy would likely remain a niche tool despite the growing popularity of Python.

7.6 Summary

This chapter presented findings from a case study of scientific software documentation crowdsourcing, based on the SciPy and NumPy Documentation Project. The analysis shows that the idea of crowdsourcing documentation developed gradually. The developers of the software did not have enough time to document their work. The software was developing quickly and the documentation could not keep up with it. At the same time there was evidence that some users produced bits of documentation for their own use. Often they then were happy to share it with the community. There was also a conviction that the users of the package are the most suitable people to write documentation. An important trigger which launched the documentation crowdsourcing process was the fact that one of the users was determined to get the documentation to a stage when any new users could relatively easily start to work with NumPy and SciPy. Getting the users to contribute required preparation. The technical infrastructure supporting collaborative work on documentation had to be set up. The format and the standard of the documentation was discussed. The discussion about the tools to use, the format and the standards, involved discussions about expectations regarding the potential contributors. The approach that prevailed was that setting up expectations which would potentially eliminate a significant number of contributors was undesirable. The documentation crowdsourcing process was supposed to be inclusive rather than exclusive. The figure below shows the different stakeholders and their roles identified in the documentation crowdsourcing process.

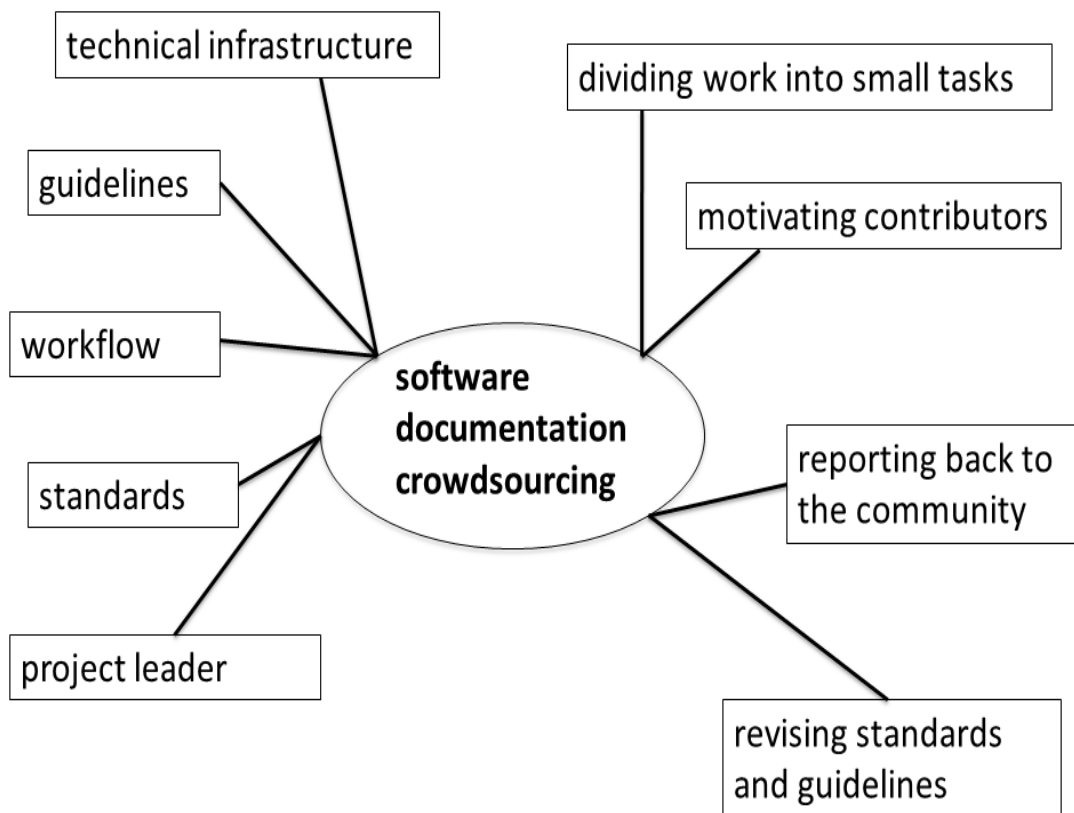
Figure 7.7: Conceptual findings from the SciPy study: stakeholders and their role in documentation crowdsourcing.



The study showed the importance of a project leader in the NumPy and SciPy documentation crowdsourcing. Apart from producing large amounts of documentation his role involved dividing tasks, managing the community's efforts, setting up milestones and keeping the volunteers motivated. Members of the community had different motivations to join the project. They found the software very useful so they

wanted to support it and improve it by writing documentation. Some, even though experienced in using it, occasionally struggled due to missing documentation. It turned out that, thanks to the crowdsourcing process, the whole SciPy community expanded. Those who had not had an opportunity to contribute to the project earlier, as they thought the only way to do it was through developing source code, were able to use their skills in other areas, such as documentation writing. Crowdsourcing improves the volume of documentation, but whether or not it improves the content remains a subject for further study. The findings provided some indication as to how crowdsourcing may be organised. They also revealed potential challenges and benefits not directly related to documentation improvement, such as community expansion. The following figure summarizes the considerations for documentation crowdsourcing.

Figure 7.8: Conceptual findings from the SciPy study: considerations for documentation crowdsourcing.

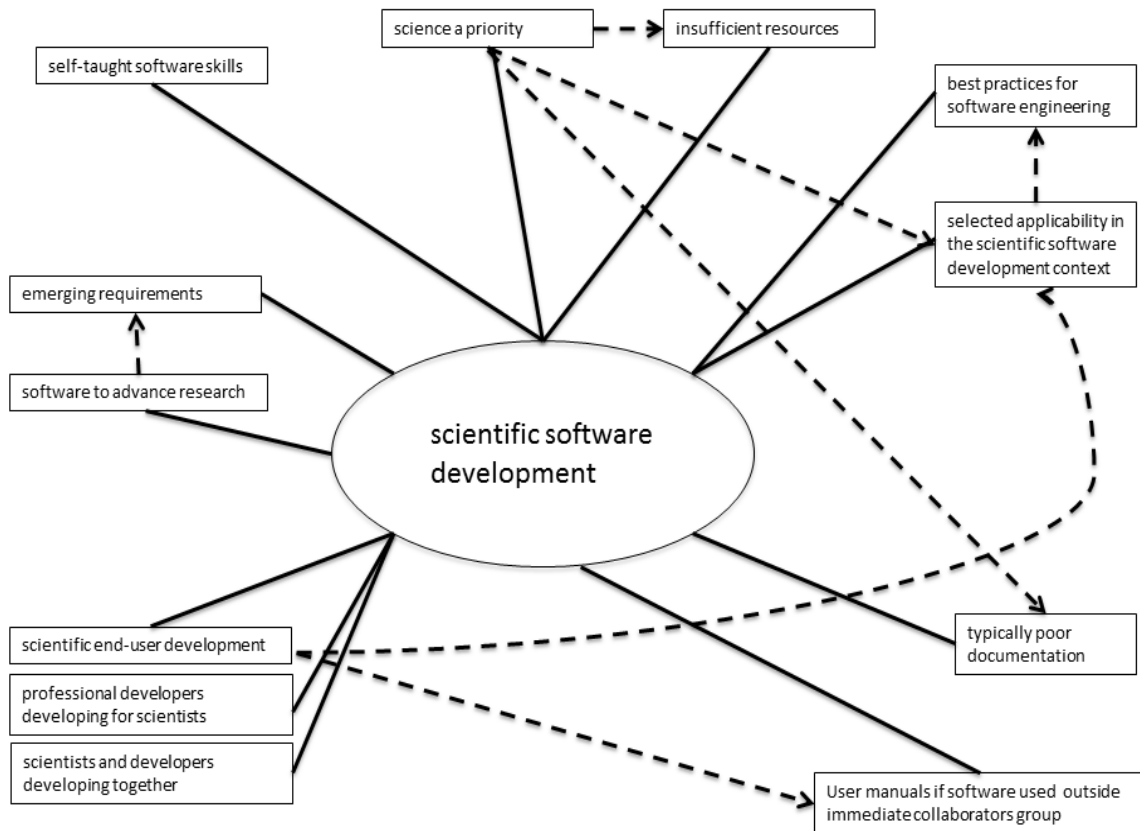


Chapter 8 Conceptual Findings and Discussion

8.1 Research summary

The research presented in this thesis explored a variety of aspects related to scientific software documentation. The preliminary study, which looked at the transition from developing scientific software in an academic environment to working on a commercial scientific software package, indicated that poor scientific software documentation is a cause of many issues. This conclusion led to the first empirical study, which explored further the role of documentation in scientific software in the change of context from scientific end-user development to developing software for others. One of the key findings from study 1 was the *role of the user community in creating documentation*. Therefore the following study looked in more detail at the ways scientific software users used documentation. The prominent finding from study 2 was that *a significant part of knowledge about the software resides within the user community*. Study 3 investigated how the knowledge that users have about software may be captured and shared. This final study looked at a particular case of harnessing the potential of the user community to write documentation through crowdsourcing. This final study connected the work done in the previous empirical studies, looking at both production and use of documentation in scientific software. In addition to helping to refine the findings from the previous empirical studies, study 3 provided a basis for forming practical guidelines for crowdsourcing software documentation. The figure below summarizes scientific software features. Some of these relate directly to documentation issues, while others provide contextual information for the issues raised in the studies. Each feature (shown in the rectangles) describes the characteristics of scientific software development and the dashed lines show relationships between some of the features (where appropriate). The priority that advancing science has is often a reason why there are insufficient resources for scientific software development. It is also one of the reasons for selected applicability of best practices in the scientific software development context and is often responsible for poor documentation. The applicability of only selected best practices is directly related to the scientific end-user context. If software is used outside of that context (and by more scientists than just the immediate collaborators group), user manuals are likely to be provided. Since scientific software development typically serves advancing the research, the requirements are constantly emerging.

Figure 8.1: Summary of scientific software features.



8.2 Emergent themes

By exploring how scientific software documentation is used and produced, this research uncovered a variety of aspects related to scientific software documentation such as:

- different documentation stakeholders and their interactions with their respective communities (discussed in section “Spectrum of users”),
- the spectrum of users stretching from end-users to user-developers (discussed in section “Spectrum of users”),
- the spectrum of scientific software, from one-off software to re-usable software (discussed in section “Spectrum of scientific software”),
- the values applied to scientific software (and documentation) production (discussed in section “Values” and “Reproducibility, reliability, and trustworthiness”),
- the main issues with scientific software documentation (discussed in section “Issues with documentation production in scientific software development”),

- the benefits of documentation production (discussed in section “(Hidden) benefits of documentation production in scientific software development”),
- that written documentation and informal knowledge exchange go hand-in-hand, to inform scientists about scientific software (discussed in section “Redefining documentation”).

Most importantly, this thesis presents empirical evidence showing that the exchange of knowledge about scientific software goes beyond formal documentation such as a user manual or comments in the source code. Much of the knowledge about the software resides within the community that clusters around that software. And these two sources, written documentation and informal knowledge exchange, go hand-in-hand, allowing scientists to use scientific software. These themes will be discussed in more detail, in turn.

8.2.1 Spectrum of users

A significant stakeholder group consists of scientific end-user developers who typically rely on limited documentation, as the expectation is that they are the only ones who will ever use the software, and that they will basically use it only once - in order to achieve their primary research goal. But these stakeholders occasionally need more elaborate and comprehensive documentation. When we look beyond the scientific end-user context, another group of scientific software documentation stakeholders are users of the scientific software developed by other scientists. The users represent a continuum: from *black-box users* (or *end-user*) who basically run the scientific software but are not interested in the details of the implementation to *white-box users* (or *user-developers*) who actively engage with the source code and develop new features or fix bugs. The former need very different documentation to the latter. Additionally, the activities of the white-box users who directly contribute to the software mean that the documentation created by the original software developers requires constant updating. Finally, the findings in this research provide an indication that other scientific software documentation stakeholder groups include various resource providers and collaborators, such as professional software developers.

8.2.1.1 Scientific end-user developers

End-user development is a common context for scientific software development; often the producer and the user of both the software and its documentation is the same person. The empirical studies reported in this thesis confirm that many scientists, at least at the beginning of their work in software, are end-user developers as reported in other relevant literature, such as (Segal, 2005a). Documentation intended for one’s own use can be very limited, because all the necessary information resides within one’s mind - as argued by the participants of Study 1 (reported in chapter 5). Arguably, producing detailed documen-

tation in the scientific end-user context can be counterproductive. Ideally, only essential information should be captured but what is essential?

Findings from the same study showed that when the scientific end-user developers had to return to their source code to make changes, they had already forgotten part or all of the necessary information. Memory is volatile, and details of implementation, solutions undertaken to “cut corners”, or things on a “to-do list” may easily fade away. Often this does not pose problems, as the software is for one-off use. However, when one needs to come back to the software, gaps in memory and the lack of documentation lead to difficulties. Moreover, as the developers understanding and perspective may change over time, so too what is essential may change.

8.2.1.2 Black-box users and white-box users

Scientist-users constitute one of the main documentation stakeholder groups. This group also bears some particular characteristics which appear to be specific to the context of scientific software and which should be taken into account when scientific software documentation is considered. Scientist-users can be represented as a continuum. On one end of this continuum there are users who simply run scientific software and do not investigate how it works. They want to know the science underpinning the software but do not need to seek any information beyond that. They can be called *black-box users*. On the opposite end of the continuum there are scientist-users who not only run the software but actively engage with its development. These *white-box users* add new features and fix existing bugs. In order to be able to do this they need to know all the nuts and bolts of the particular piece of software.

In practice, the distinction between these two groups of users may be unclear. Any individual moves up and down this spectrum depending on the software, and the particular task at hand. Some black-box users may actually want to know a few implementation details of the software. Some may occasionally fix a bug or engage in developing a new functionality. And a black-box user in one context can be a white-box user in another. Hence it is more accurate to say that scientist-users can be represented as a continuum rather than several discrete groups. How then should scientific software documentation reflect this continuum of users?

Moreover, white-box users’ contributions call for updates in documentation. New functionalities need to be described so that the black-box users are able to use them and any other scientist-developers are able to maintain the added source code. Solutions to fixing bugs need to be documented. The original developers of the particular piece of scientific software are unable to predict what the white-box users will contribute and therefore are unable to provide the relevant documentation.

8.2.1.3 Other stakeholders: resource providers and collaborators

There are other stakeholder groups in scientific software documentation. Even though these groups were not the main focus of the empirical studies reported in this thesis (as the main focus was scientist-developers and scientist-users), they should still be considered in the discussion. The first group is the resource providers. These may be administrators of computing resources on which scientific software is installed and run, as well as decision makers who decide about grants and evaluate research projects. The second group is collaborators such as professional software developers who may be working in a research project consulting on the development of scientific software (as in Segal 2009 and Easterbrook and Johns 2009). The stakeholders from these groups were occasionally mentioned by the participants of the empirical studies. For example, in the study on scientific software documentation use, the interviewees mentioned consulting the administrators of the computing resources which they used. These administrators typically did not have background in a given scientific domain and were responsible for maintaining the infrastructure which the scientists used. They did not need to know the science underpinning the software package, but they needed to understand the softwares technical requirements (for example, the libraries).

All of these stakeholder groups need different information about scientific software. They seek particular facts and want to be informed in a particular way. Whereas professional software developers and computing resource providers are likely to be interested in the technical details, the decision makers are probably not interested in technical aspects of the software but rather its impact on the research. The former are most likely to look for installation and configuration guidelines, while the latter may seek implementation details. Theoretically this means that several sets of scientific software documentation are needed. Considering how problematic it is already to provide sufficient documentation, addressing the perspectives of different stakeholders may be an issue. Maybe it would be possible to reuse existing documentation for this purpose. Information captured about the science related to the software may be useful for the decision makers. Instructions for users (both white-box and black-box users) may include at least some of the details sought by the computing resource providers and professional software developers. With consideration of the different perspectives, it may be possible to produce documentation which could be useful for more than just one group of stakeholders.

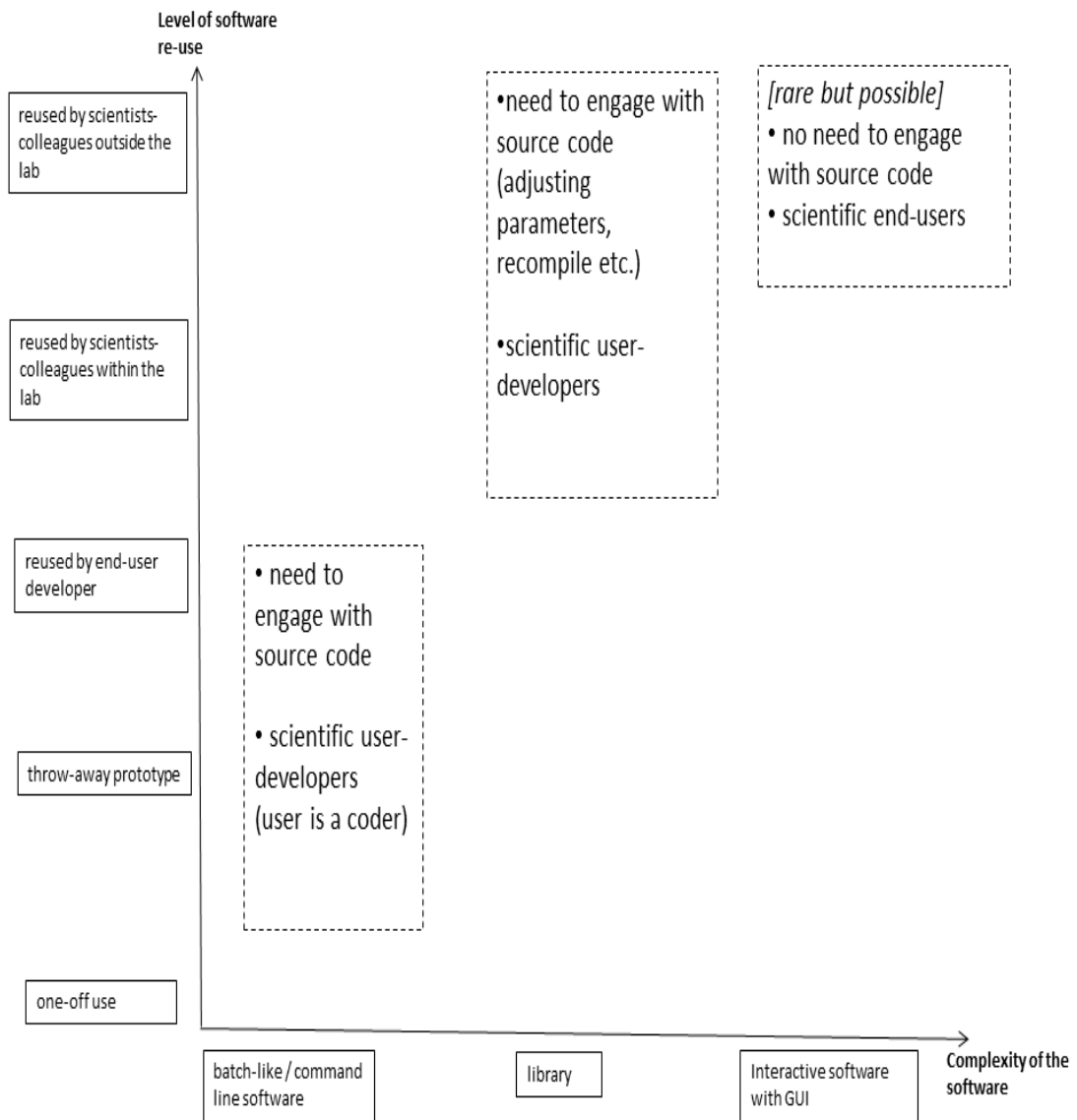
8.2.2 Spectrum of scientific software

Similarly, there is a spectrum of scientific software, from disposable programs intended for one-off use, to software shared with others. The software referred to by the participants of the studies about documentation production and use (reported respectively in Chapters 5 and 6) varied from short batch programs (a few hundred lines of code) to large software suites with a GUI and software libraries. The

algorithmic complexity of the software also varied from highly advanced code implementing numerical methods or using complicated parallelisation techniques, to less complex pipeline processing code. Investment in documentation is unlikely for one-off programs for ones own use, but documentation for software to be used, edited, and extended by others is essential, particularly as the user population diversifies. Just as scientist developers may change roles over time, so software also has a life of its own, sometimes developing from disposable to re-usable, from a specific context to more general use, from academic to commercial production.

The context of re-use may also change, from within-group, to between-groups, to more general or open source sharing. Each transition between contexts has implications for documentation, particularly as the distance between the developers and the users increases. This is particularly important for scientific software. Unlike other commercial software, which is often intended as “press a button and run” applications, scientific software usually requires technical engagement even from black box users, for example compiling code, changing processing for different data sets or calling code from elsewhere. In effect, to be a user is to be at least a rudimentary programmer, with the accompanying implications for documentation that must accommodate use, re-use, and re-coding. The figure below visualizes the spectrum of scientific software with regards to its complexity and level of reuse. The vertical axis represent the different levels of the software reuse. At the bottom, it is “one-off use” when software is used once to obtain the desired results and then abandoned (typically in the scientific end-user context). The throw-away prototype is likely to be revisited and changed but it is usually short-lived and unlikely to be reused. Then there is software that actually is reused by (gradually) expanding user-base: from the user-developer to a potentially wide group outside the original lab. The horizontal axis represents the complexity of the software from a batch-like application (typically without a GUI), through a library to an interactive piece of software, potentially with some kind of GUI. The software that is not for reuse usually requires direct engagement with the source code and does not come with features such as interactivity. Libraries and interactive software (potentially with GUI) are developed with reusability in mind. Libraries are intended to be used by programmers whilst interactive software, especially when it comes with GUI, not necessarily.

Figure 8.2: Conceptual findings: Scientific software spectrum.



8.2.3 Values

Scientific software has two masters: *science* and *software*. For many, science remains the top priority, and software development remains secondary to the scientific goal, with little dedicated investment or attention to practice. Their priorities might be expressed in a simple hierarchy:

1. doing science: advancing one's own research remains the centre of attention, and all efforts are focused on obtaining the scientific results, and then on publishing them;
2. creating software: software development remains a part of the research process and serves primarily if not only to obtain the scientific results;
3. producing documentation: writing documentation directly benefits neither advancing research nor developing software, and documentation is produced only when it is unavoidable.

This has emerged from the empirical studies presented here, as well as from the literature (for example, (Howison and Herbsleb, 2011)). Therefore, scientist-developers values for their software are shaped by the need to support their science, emphasising rigour/reproducibility, re-use (bringing the potential for efficiency in the conduct of the science), and reputation (given that the rewards in science are based on scientific findings and publication). Investment in documentation will rely on the perception by key stakeholders that science will benefit from good documentation, through improvements in rigour, reproducibility, and re-usability. Such recognition may often come only as a reflection when a given project is finished and the issues related to poor documentation come to light. The benefits of documentation production are often noted only post-factum. Therefore the shift in the values attributed to it may also only come with time and experience.

8.2.4 Reproducibility, reliability, and trustworthiness

Reproducibility is fundamental to science, and is therefore a key concern for scientific software. Being able to re-apply software to reproduce results depends on the quality of both the software and the documentation, as re-application requires understanding of the science and assumptions underpinning the software, as well as its correct use.

Related to reproducibility (and equally crucial) are reliability and trustworthiness. Reliability is related to the consistency of, for example, the quality or availability of something. Trust is the matter of believing in reliability. If the scientists cannot trust the software to be correct and to do what it is expected to do, they cannot use it for their research. Unreliable software means unreliable science. If a user finds a particular piece of software to be unreliable, they are unlikely to report this in a publication (Howison and Herbsleb, 2011). Therefore the publications are not source from which scientists could learn about problematic software. Although, as it is discussed further, the publications may be used as a source for recommendations for reliable software.

In general, testing scientific software in order to ensure reproducibility and reliability is a challenge. Hook and Kelly (2009) suggest testing scientific software at three different levels: scientific validation, algorithm verification, and code scrutinization. This is possible for a scientist-developer who has access to the source code, who understands it and is able to engage with it. Many, if not most, researchers do not have this luxury. The findings from the study of the scientific software use (reported in chapter 6) indicate that researchers find their peers' recommendations with regard to the software trustworthiness to be valuable. One of the participants of the study reported finding new pieces of software by looking at scientific publications reporting software use. It appears that the strategy was: if a particular paper was published in a peer-reviewed journal and in order to obtain the results in this paper, a particular piece of software was used, it was an indication that this software should be trustworthy. The scientific software

community becomes a kind of guarantor for the software trustworthiness. The community's knowledge captured in different formats constitutes a part of the software documentation.

It is possible that, if scientists rely on their peers' evaluation of scientific software, they may trust their peers in other matters related to software use and development. The findings from the study of scientific software documentation use revealed that if the documentation provided with the software failed to give a solution, the scientist-users consulted other software users, either face-to-face or online. There may be also information useful for scientist-developers who may be trying to develop a piece of existing software or look for programming hints. Occasionally, these scientists may look for information beyond the scientific community, e.g. a general programming forum for Python. One of the interviewees in the study of scientific software use during the work-based interview used a range of generic resources with advice on software development. The question is whether and to what extent the information about the software coming from the scientific community should be questioned.

8.3 Issues with documentation production in scientific software development

Each empirical study reported in this thesis revealed issues with documentation in scientific software development. The outcomes of the preliminary study showed that problems with documentation may lead to difficulties with software maintenance and use. The three main studies not only confirmed that but provided a better insight into the issues and why they occur. This thesis does not claim that the problems with software documentation are exclusive to scientific software development. But the consequences of poor software documentation may be exacerbated by the scientific software context. For example, reproducibility of research performed with a piece of poorly documented scientific software may simply be impossible. Poor documentation may also lead to unwitting misuse of the software, and this may lead to incorrect results. This dissertation contributes to understanding the reasons why the issues with software documentation occur in scientific software development. The findings discussed in this section contribute to answering the research question *How is scientific software documentation produced?*.

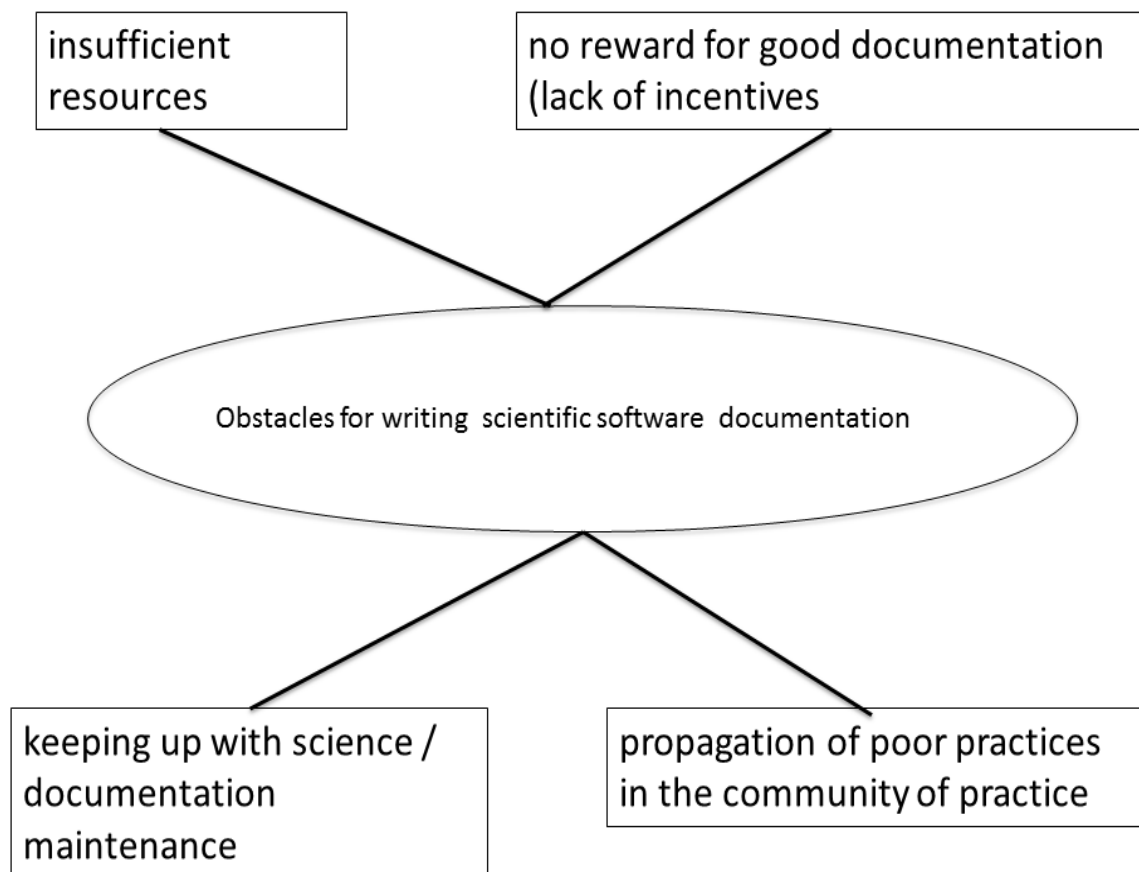
The empirical evidence points out to the following issues with documentation production:

- insufficient resources allocated for documentation production,
- lack of incentives for researchers to produce documentation,
- possible proliferation of bad practices through uncritical reliance on advice from the community,
- the difficulties in keeping up with the constant advances of science.

Addressing these issues is challenging. At the same time the findings indicate that by using the knowledge residing within the community, scientists are able to deal with the shortcomings of the documen-

tation. And documentation crowdsourcing may be a way to help addressing the aforementioned issues. The figure below summarizes the obstacles for writing documentation.

Figure 8.3: Conceptual findings: Obstacles for writing documentation



8.3.1 Insufficient resources

Lack of funding and time appear to be one of the reasons for not producing documentation. When there is not enough of these resources to develop robust scientific software, the production of the relevant documentation is also affected. Especially in the scientific-end user context when the main goal is advancing the science, these resources tend to be limited. Time is too precious and hence cannot be spent on things which are not essential. When scientists set up their priorities, scientific software documentation production tends to be pushed down the line. In professional software development, documentation may not be given extensive attention (for example, Lethbridge et al. 2003). But in professional software development documenting is a part of the development process (Rettig, 1991) and it is highly likely that there are resources allocated for producing documentation.

The scientists *do* recognize that documentation is important, but at the same time they have other more urgent tasks to complete. Segal's (2007) studies show that scientific end-users treat software develop-

ment as a means to an end. In her studies the participants used the source code as documentation. The studies reported in this thesis support that. However, they also suggest that if scientists had more time, at least some of them might consider spending it on documentation writing. In particular, those scientists who experienced how painful the lack of documentation may be are likely to do it. Such problematic experiences happen often when the scientists move from scientific end-user development to developing software for a wider community. Yet still the resources which could potentially enable them to produce documentation are limited. Scientific software development is not a source of recognition in research (Howison and Herbsleb, 2011). For that reason there are problems in allocating sufficient resources for software development and all related activities, such as documentation production. It may also be that resources should be allocated specifically for producing documentation, as in the case of the SciPy community. As the participants of this study (reported in chapter 7) reflected, the fact that there was funding dedicated to the documentation effort and a community champion whose full-time job was to work on documentation was crucial for the whole project. It was made clear to everyone that these resources were for documentation production and hence could not be shifted to support other tasks.

8.3.2 Lack of incentives for researchers to produce documentation

Putting extra effort into software documentation writing typically does not benefit a researcher's career. Academic reputation and careers are built on scientific publications (Howison and Herbsleb, 2011). It is possible that when a piece of scientific software is published and released, a publication about that software is highly cited (for example, Hull et al. 2006). But the further maintenance of the software, although it requires work and resources, is not a source of further publications. Scientists do recognize the value of documentation, but they are conscious that the investment they would make in producing it will not pay off. It may even hinder their careers, taking time from their scientific research.

Moreover, documentation production is often perceived as a dull and boring task. The findings of the preliminary study (reported in chapter 2) as well as the study of the scientific software documentation production (reported in chapter 5) showed that writing documentation was something which the scientists did not enjoy. Documentation writing may be appealing to some scientists but that is a rare occurrence. Only one interviewee in the study of the documentation production explicitly said that he liked documenting his software. In professional software development, documentation production may be perceived as a mundane task. On the other hand, there is a career path or at least a set of job posts for those writing documentation full-time. For example, technical writers are employed in many software companies and their job can be highly valued. When addressing issues related to scientific software documentation one needs to consider the context of scientific software development. The primary goal is advancing research and software development mainly serves this purpose. There is little or no formal

recognition for documentation production. One of the interviewees from the study of scientific software documentation crowdsourcing stated clearly that putting effort into documenting scientific software does not help in developing research careers.

8.3.3 Keeping up with science

Scientific software supports research and as the latter advances, the former has to keep up. As Dubois (2005, p. 80) says: “For most scientific programs, the rate of change doesn’t decrease significantly even after many years. Like sharks, scientific programs that aren’t moving are dead”. These movements are typically due to advances in science. As the results of the study of scientific software documentation use (reported in chapter 6) showed, the users of scientific software may want to alter it to a lesser or greater extent, in order to accommodate some new scientific ideas. The findings from the study of scientific software documentation production (reported in chapter 5) reveal that even in the scientific end-user context, when software is expected to be one-off use, the scientist-developers may need to return to it and change some parts of the source code as, their research advances. The directions in which the science will change are difficult if not impossible to predict. Therefore any changes to the scientific software resulting from advances of science cannot be pre-documented. It is not only scientific software that has to keep up with the advances in science, but also the software documentation. Maintaining consistency between research and documentation may be a challenge in producing scientific software documentation.

However, one aspect of scientific software documentation may be able to keep up with the advances in science. The publications reporting the research results obtained by using the scientific software can document the science underpinning the source code. As the scientific model evolves, new papers are published. These papers record and describe the evolution of the science. Even if these papers are not related to the particular piece of software, they may still be of use as the documentation of the aforementioned science. The data collected during the study of scientific software documentation use showed an example in which a scientist, using a paper discussing some advancements in the particular research area he worked in, developed software which not only allowed him to reproduce the reported results but also to move on with his own work and his own source code. Scientist-developers commonly document the science behind the software (Sanders and Kelly, 2008).

It should also be noted that keeping the documentation up to date is not a problem specific to the context of the scientific software development. Professional software engineers also “rarely update the documents”, apart from the documentation used for testing and quality (Lethbridge et al., 2003). The software changes because there might be new features added or bugs fixed. In non-scientific software development these new features or bugs may be difficult to predict, just like in scientific software de-

velopment. One of the approaches to ease software documentation updating is using tools supporting automatic documentation generation from the source code. One of the participants of the preliminary study (chapter 2) reported using such tools and being able to update the documentation as his software was changing. The case of crowdsourcing the production of SciPy and NumPy docstrings described in chapter 7 indicates that it may be possible to keep the documentation embedded in the source code up to date thanks to community efforts. It may be that some of the approaches used in the context of commercial software development may possibly be applicable and useful in the context of scientific software development.

8.3.4 A community of practice - difficulties in changing practices

Scientist-developers learn what information to capture about scientific software and how to capture it by looking at existing documentation. Effectively, they often learn from their colleagues by following their direct advice or expanding documentation which is already in place in the scientific software project. This exchange of knowledge and experience and cultivation of practices related to documentation production appears to fit in well within the notion of community of practice (Wenger, 2007) or a network of practice (Brown and Duguid, 2002) if they are not physically co-located but rather communicate primarily via the internet. Both communities and networks of practice, as discussed by Wenger (2007) and Brown and Duguid (2002), can be very effective in disseminating knowledge and assisting their members in furthering their skills. Scientists who use and develop a particular piece of scientific software (or a software suite or a set of applications) appear to form communities and networks of practice. The community and network of practice which the scientist-developers form can be a place to learn how to produce documentation. In the study of the scientific software documentation production (reported in chapter 5) one of the participants explicitly said that when he produced documentation, he continued the documentation produced earlier by one of his colleagues.

Being embedded in a community of practice may mean that the members of the community do not notice if their practices need any alterations. In the case of the scientific end-user developers who move towards developing for a wide community of users strongly rooted documentation production practices may be difficult to change. In the context of scientific end-user development documentation is mainly needed for the scientist-developer himself. Often there may be no other developers than the scientific end-user developer himself and in such cases documentation for external people interested in development is not necessary. This argument was brought up by the participants of the preliminary study (reported in chapter 2) and the study of scientific software documentation production (reported in chapter 5). Whilst scientist-developers acknowledge the importance of the documentation when the context of the scientific software development changes, their habits of producing documentation may remain

so strong that they may not change with the changing context. For example, documentation for other developers may still be scarce. Being a part of a community or a network of practice may contribute to this issue, as within the community of practice one of the common practices may be producing scarce documentation for other developers. It may not occur to members of this community, that some of the practices may need to be revised.

8.4 (Hidden) benefits of documentation production in scientific software development

The thesis also identifies the benefits of scientific software documentation. Documentation production aids reasoning. Writing documentation allows the documentation's author to reflect on the software design, implementation details and solutions. In the case of scientific software, this process may also spur new ideas related to scientific research. Another benefit is that scientific software documentation may support reproducibility of scientific results. Finally, engaging the users in documentation production may help to expand the community of the software developers. That is, by writing documentation the users may gain confidence, skills and further interest in contributing to the software source code.

There are more benefits to writing documentation than it may seem at a first glance. The process of documentation production may facilitate reflecting on one's own work. It encourages the scientist-developers to reconsider a number of aspects of the software which they created. It may lead to making improvements in the software itself. Sometimes it may inspire new research ideas or bring to attention any issues which might have been unnoticed before. The indication that documentation production aids reasoning was present in particular in the first main study reported in chapter 5. It was later explicitly discussed by the main NumPy developer who wrote "Guide to NumPy" whilst he was still working on the library. Putting down in writing the guidelines for usage and the functionality of the library modules made him think what he really wanted the software to do. This influenced his work further as a developer. It would be interesting to investigate further how exactly documentation writing aids reasoning about software development and the research related.

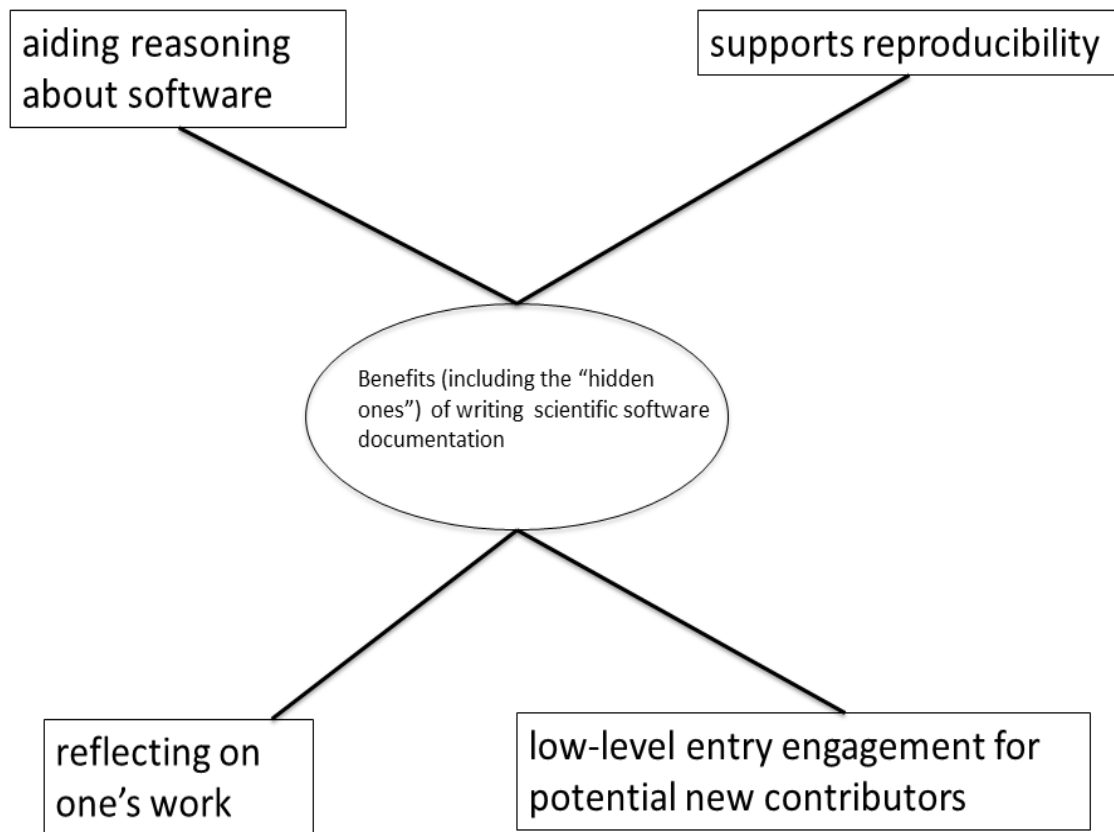
Another benefit of documentation production which may potentially be very appealing to scientist-developers is the contribution that documentation makes to reproducibility. Lack of documentation is likely to lead to a situation in which it is impossible to understand how a scientific model was implemented. The inability to verify or to reproduce the reported scientific results may make the researchers distrust the reported research (Hook and Kelly, 2009). Original scientist-developers may also need to come back to their source code and re-run calculations or simulations. Poor or lack of documentation makes this task extremely difficult, as the results of the study 1 showed. In some extreme cases, as

indicated by the findings of the study on scientific software documentation use (reported in chapter 6), scarce scientific software documentation may lead to the scientist spending several frustrating months focused almost entirely on trying to understand how the specific method was implemented. If the software is documented, there is a good chance that the captured information will allow for reproducibility. Producing scientific software documentation may contribute to the improved reproducibility of science. It may be worth investigating what information needs to be captured about the software to ensure that the research which it underpins is reproducible. It may vary between disciplines and depend on a variety of factors related to reproducing research results. Certainly better understanding how scientific software documentation can support reproducibility would be desirable.

Finally, engaging with documentation production may encourage scientist-users to become scientist-developers. The study on the SciPy and NumPy documentation crowdsourcing project (reported in chapter 7) provides evidence that documentation production may be an enabler for those members of the community who have not previously been a part of the development team to join in development activities. It may not be very common, and it is probably likely to happen to users who are already using the software a lot and feel a part of the user community. From the community perspective this may be very beneficial. The project gains not just a developer but a developer who is already known to other community members, who is familiar with the software, and who has already invested a considerable amount of effort and time into the project, and hence may be less likely to abandon it quickly. There is however a caveat to scientist-users turning into scientist-developers: the more they engage in software development, the less time may possibly be left for them to focus on their scientific careers. This point was brought up by one of the interviewees in the SciPy and NumPy documentation crowdsourcing study. Considering that literature (for example, Segal 2007) shows that for scientists the main goal is advancing science, putting more time into software development may distract the scientists from their main goal. From this point of view, this hidden benefit of producing documentation may actually be a disadvantage.

The figure below summarizes the benefits emerging from writing documentation.

Figure 8.4: Conceptual findings: Benefits from writing documentation.



8.5 Redefining documentation

The insights presented in this thesis suggest that the definition of software documentation presented in chapter 1 is too narrow to cover all the ways information is exchanged about software, and that a broader definition should be adopted, or at least that a distinction should be made between two types of documentation, formal documentation (as defined in chapter 1), and informal documentation, supported through dialogue in the community.

8.5.1 Formal and “informal” documentation

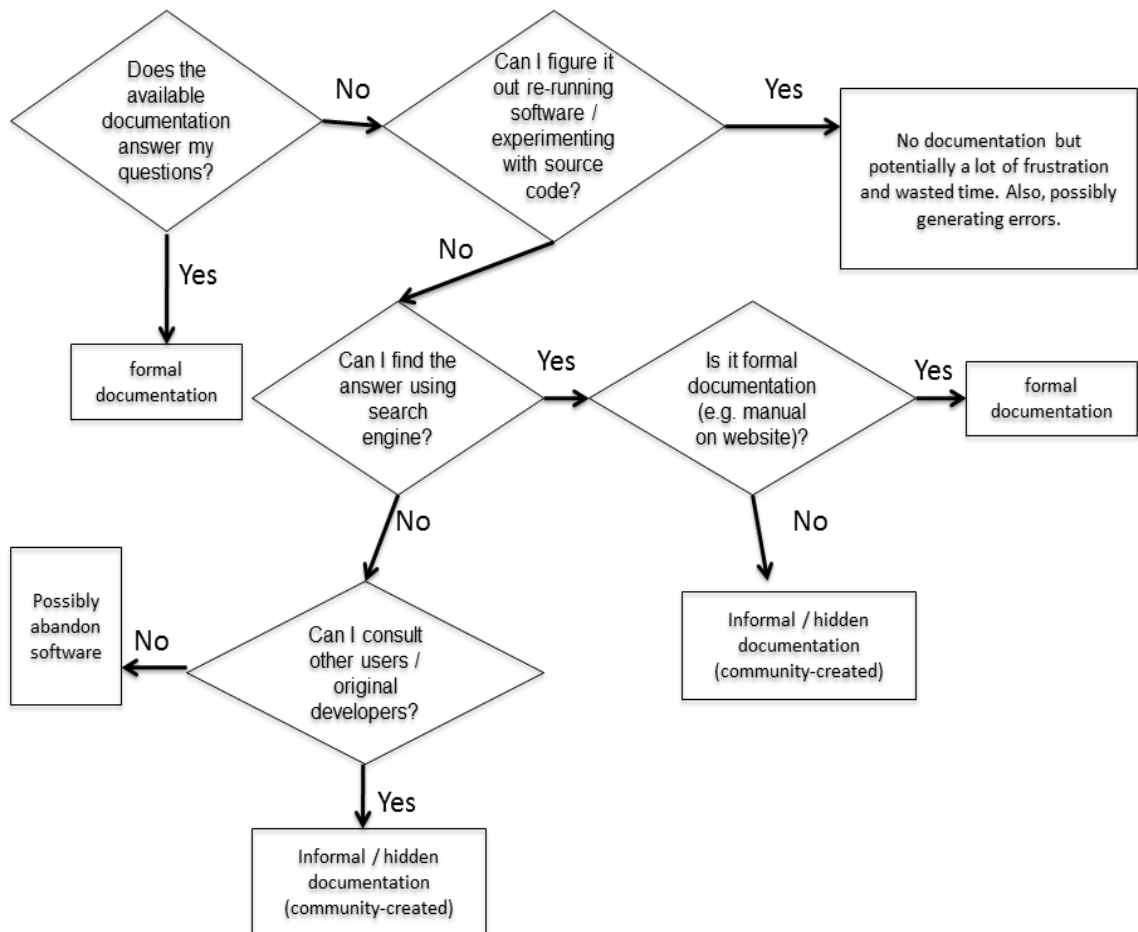
Formal documentation is information about software captured in writing and intended to stand alone. Typically, written documentation has structure and covers a certain set of aspects of the software (for example, how to install and use the software). Although the documentation may be incomplete, the original aim of its author was that it would stand alone to provide necessary information about the software. Formal documentation may be a user manual (in an electronic or a paper form), a wiki, comments in the source code and the source code itself.

The second type of documentation is much more “informal”; it is not in the form of comprehensive and

formatted documents. It could also be called “hidden” because it is not apparent at the first glance, but is based in dynamic knowledge sharing in the user/developer community. This knowledge sharing can be via an internet forum or at a research conference. If it is the former, it is still captured and, if the forum is archived on the server, can be accessed in the future. If it is the latter, then it is volatile, and the shared knowledge is less likely to be passed on to other community members. Nevertheless, the findings from this PhD research indicate that, even though volatile, this kind of documentation is highly valued by the scientists. It may be partially due to the fact that the answers which they receive when talking to other researchers are well targeted and provide solutions to problems which they are experiencing.

This redefinition emerges from evidence from the scientific software context. However, it may be relevant in other contexts as well. For example, the evident role of the user community in sharing knowledge about the software and filling in the gaps when documentation is missing may also be present in other domains. This is a matter for further study. The figure below depicts a workflow of seeking information about scientific software. It shows in which cases a user or a developer is likely to arrive to using formal or “informal” documentation.

Figure 8.5: Conceptual findings: Pattern of using formal/informal documentation



8.5.2 Different roles of documentation

In the scientific software development context documentation can have different roles.

- It provides information for the users on how to install and run the software.
- Documentation may discuss the science underpinning a given piece of software. It explains the scientific model and, ideally, the details of its implementation. Such documentation may be in a form of a research paper or a collection of research papers. The reference to them may be in the user manual, README file, on the website or included in the source code comments.
- Documentation may describe how the code works, for example giving an overview of the process structure or data flow, capturing assumptions or constraints embodied in the implementation (for example, explaining how the scientific algorithm has been interpreted may have implications for its application), or describing the data requirements. This complements the previous role (documenting the underpinning science) by covering how science may be conducted using the software (i.e., documenting how the science is enabled by the software).
- Documentation is also a way of sharing knowledge within the community. Both the user and the developer communities exchange knowledge and experiences about the software via the documentation. It is typically the “hidden documentation” discussed in the previous subsection. Users and developers share their knowledge and experiences through conversations, for example via a mailing list or a forum or during discussions online or face-to-face.
- Documentation also helps in establishing trustworthiness in the given piece of software.

8.6 Considerations for crowdsourcing documentation

The SciPy documentation study provided sufficient evidence for forming guidelines that are useful to take into account when considering crowdsourcing documentation. Crowdsourcing may combat some of the obstacles to producing documentation (as outlined in chapter X), and producing documentation in this way has added benefits (such as building a community of practice). The matters to consider for documentation crowdsourcing listed below are complementary to the conceptual findings discussed earlier in this chapter. These provide a practical dimension to the concepts that emerged from the data.

- Technical infrastructure should support both the community members who want to contribute and the community champions who manage the collaborative effort. The infrastructure may reinforce the inclusive or exclusive nature of the crowdsourcing project. The simpler the infrastructure is to use, the more people are likely to use it and get involved. On the other hand, if use of the infrastructure requires knowledge of some specific tools and skills, it may help to filter the community for the desired contributors who have the selected skill set.

- Stylistic guidelines help with writing consistent documentation enhancing its usability. The guidelines should clearly state what to do and how to do it. For software documentation, the guidelines should focus specifically on the information that needs to be captured, the level of detail, the language, the order of information, the format and so on. The stylistic guidelines should be readily available for anyone wanting to contribute.

Clear and sustainable workflow turns crowdsourcing from a chaotic process into a process resembling a system of gears that move together with precision. The workflow can help arrange the tasks that are involved in documentation into a sequence that prevents clashes and deadlocks. The workflow help new contributors joining the project to get up to speed.

- The project leader coordinates the process of crowdsourcing. The leaders role is not limited to executing the workflow. The coordinator may need to use a divide and conquer approach, breaking larger and more complex goals into self-contained tasks, which then are assigned to the community members. The leader may need to set up and revise the milestones. One of the most challenging responsibilities may be keeping up the momentum, ensuring that, when the novelty and enthusiasm wears out, the collaborative effort does not die down.
- Motivation and recognition is much needed to keep the project running. Showing the contributors the outcomes of their effort helps them believe that what they do makes sense and has value. It is about making the return of investment visible. Regular updates on the progress and tokens of recognition show the contributors that their investment is not wasted and does not go unnoticed.

8.7 Significance of findings

The findings of this PhD study are significant from both researchers' and practitioners' perspectives. The former may use the results as a basis to define foci for their studies and the latter will see that documentation has significant benefits. The findings from this PhD research contribute to the knowledge about the practices of scientific software development. These findings not only contribute to filling in the gaps in the existing literature but also inform directions for further research (as discussed it is discussed in the final concluding chapter). For practitioners, the findings of this PhD study may have direct implications. This may then have implications for the potential success of the given scientific software development project. Understanding the needs of different stakeholders in the project is without doubt one of the keys to successful collaboration. The findings presented in this thesis should not be treated as prescriptive but rather indicative or highlighting the possible actions which may be undertaken by the practitioners and which, eventually may make a contribution to their projects.

The significance of these findings is emphasized by the fact that they all emerge from empirical evidence. The data collected during the studies consisted of 42 interviews, archives spanning 12-years

of three mailing lists, logs from the documentation infrastructure server, two progress reports and one technical overview. The studies covered several different scientific domains (including physics, computational chemistry, applied mathematics, biological sciences) looking at experiences of scientists at different stages of their career.

Chapter 9 Conclusion

9.1 Main findings and answers to the research questions

As software becomes a commonly used instrument in science, attention is needed to the practices associated with scientific software development. Just as there is good understanding of how physical laboratory equipment and procedures should be handled (such as calibration and maintenance), there is a need for good understanding of practices and approaches to scientific software development and use. Documentation is one of those practices which in the scientific context gains particular importance, because it stands at the interface between the software *per se* and both the science that underpins it and the science that that software enables.

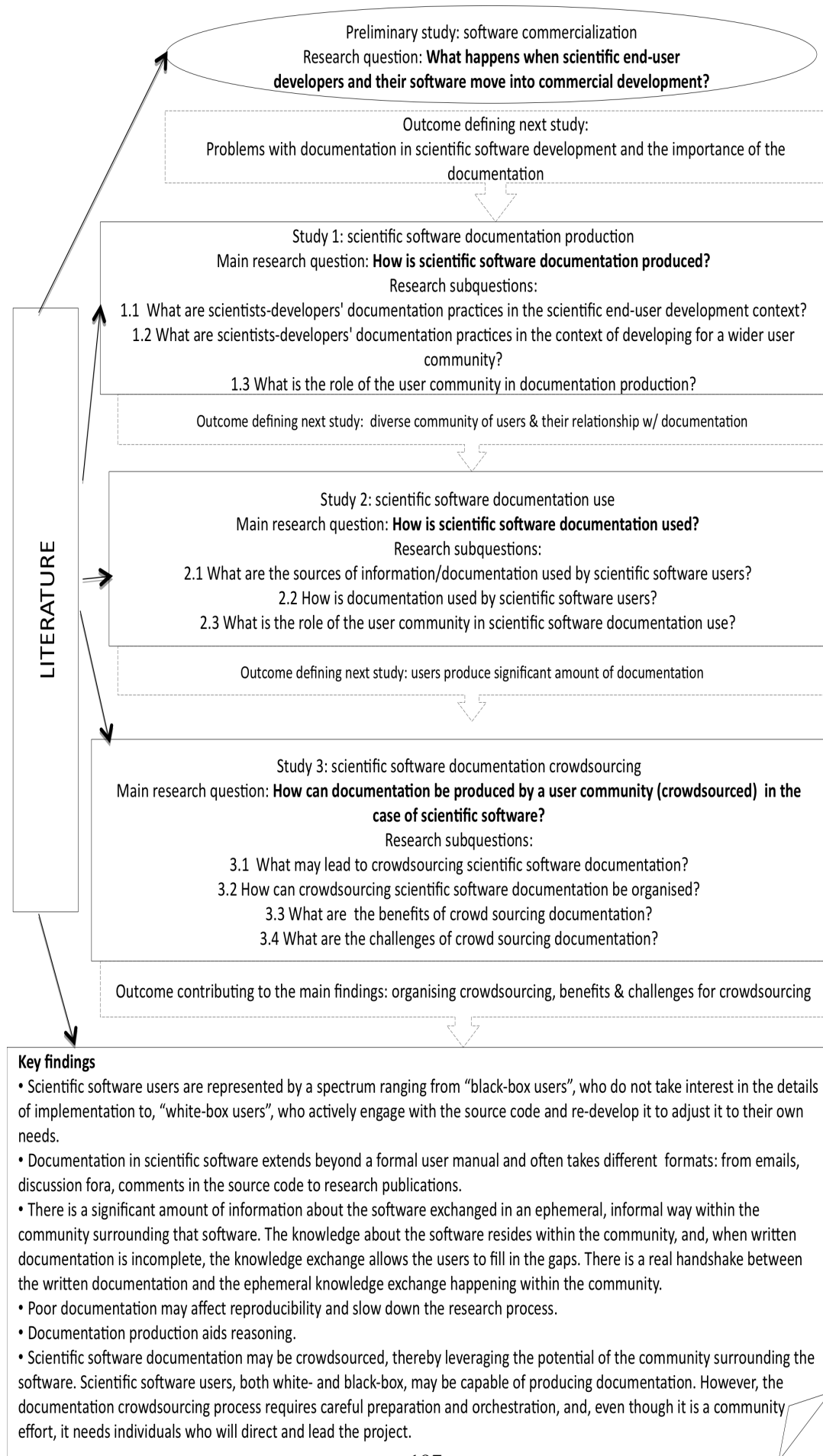
Documentation is one of the main ways of communicating information about software. Effective documentation communicates both standard operating information about what the software does and how to use it, and, importantly, information about *what may not be explicit* in the source code: the interpretation of the domain knowledge, the intentions of the developer and the assumptions associated with the code. Furthermore, in the scientific context, the need for information is not just about implementation detail but also about the method underlying the software. Documentation affects scientists' ability to reuse and maintain software effectively. Poor documentation may affect not only use and development of software, but ultimately also the science itself, in the worst case introducing flaws in the research.

The research presented in this dissertation has examined the role, production and use of documentation in scientific software development through a sequence of four empirical studies. The preliminary study examined the transition from scientific end-user development to commercialization. Its outcomes highlighted the importance of documentation and provided the focus for the rest of the research. Each subsequent study examined documentation from a different perspective: documentation production by scientist developers; documentation use and crowdsourcing documentation within a community. The structure of the research was iterative: the outcomes of each study informed the focus of the next. This structure is presented in Figure 9.1, which shows the progression, in terms both of research questions and key findings.

Tables 9.1, 9.2 and 9.3 set out the research questions and findings associated with each of the three

empirical studies of documentation, and thereby present a summary of the contributions made by this research.

Figure 9.1: Research structure revisited.



Study 1 - Main RQ: How is scientific software documentation produced?		
Research Subquestion	Findings	Chapter
1.1. What are scientist-developers' documentation practices in the scientific end-user development context?	<ul style="list-style-type: none"> - The scientific model underpinning the software is documented. - Implementation details are documented to a limited extent - just enough for the scientific end-user developer to be able to work with the software in order to progress his research. - Software documentation is given low priority, as the main goal remains the advancement of science. Software development, let alone its documentation, is secondary to the science. - User manuals are almost never produced. - In general, documentation in scientific end-user development is very limited. - Documentation is often not produced due to the lack of incentives and resources. - Documentation is captured in variety of formats. 	Chapter 5
1.2 What are scientists-developers' documentation practices in the context of developing for a wider user community?	<ul style="list-style-type: none"> - User manuals are provided. - Documentation for other developers is usually provided but often in a limited scope. - Scientist-developers learn how to document their software from their peers; the community of practice plays an important role in shaping documentation. - Documentation reflects assumptions about the user community: these assumptions are related to the level of scientific knowledge and the knowledge and skills in computing. 	Chapter 5 Chapter 6 provided supportive findings.
1.3 What is the role of the user community in documentation production?	<ul style="list-style-type: none"> - Users' feedback may impact documentation production directly, as users may be coming back to the original developers with questions about the software that may spur changes in documentation. - Developers who are past (or sometimes current) users of the scientific software which they develop, and this is often why they feel that they have the license to make assumptions about other users' knowledge and skills. - Information about software resides within the user community. Users may produce documentation. - Scientific software documentation may be produced via crowdsourcing, thereby leveraging the community's potential for exchanging knowledge about the software. 	Chapter 5 Chapter 6 Chapter 7 provided supportive findings.

Table 9.1: Study 1. Summary of findings mapped to the research questions

Study 2 - Main RQ: How is scientific software documentation used?		
Research Subquestion	Findings	Chapter
2.1. What are the sources of information/documentation used by scientific software users?	<ul style="list-style-type: none"> - End-users (black-box users) start from reading manuals and then consult other resources, typically when the manuals lack information that users need. - In some cases, information about software is provided by dedicated helpdesks and workshops and materials are often shared online. Users appreciate the helpdesk support and find it very effective. - Research publications related to the software are used by both black-box and white-box users. - Generic information about programming available online is often used by scientific user-developers. 	Chapter 6
2.2 How is documentation used by scientific software users?	<ul style="list-style-type: none"> - User manuals (if available) are initially followed by users quite meticulously. - As users gain more experience with the software the user manuals are used mainly for reference. - Online resources are used in an ad-hoc manner, typically via searching for information using keywords in a search engine. - Consulting others about the software is done only after all sources of written documentation have failed. 	Chapter 6
2.3 What is the role of the user community in scientific software documentation use?	<ul style="list-style-type: none"> - When written sources of documentation fail, the user community often is the source of information about software. Knowledge exchange happening in the user community is complementary to use of written documentation. - The user community recommends documentation resources. 	Chapter 6 Chapter 7 provided supportive findings.

Table 9.2: Study 2. Summary of findings mapped to the research questions

Study 3 - Main RQ: How can documentation be produced by a user community in the case of scientific software?		
Research Subquestion	Findings	Chapter
3.1. What may lead to crowdsourcing scientific software documentation?	<ul style="list-style-type: none"> - Incomplete or missing documentation is the main reason for crowdsourcing. - User community members are keen to share bits of documentation produced for their own use. - Individuals' needs to improve documentation are a strong driving factor for engaging in crowdsourcing. 	Chapter 7
3.2. How can crowdsourcing scientific software documentation be organised?	<ul style="list-style-type: none"> - Technical infrastructure supporting collaborative work and enabling management need to be in place. - Technical infrastructure may be used for screening the types of users who contribute to documentation crowdsourcing. - Task division, setting goals and planning are necessary. 	Chapter 7
3.3. What are the benefits of crowdsourcing documentation?	<ul style="list-style-type: none"> - Improvement of the documentation is quite likely to happen, but it is not guaranteed. - Standards for documentation can be established and endorsed. - The community around the software, of both users and developers, may expand. If documentation actually improves, the software may become more attractive to new users, whilst the users who contributed to the documentation may gradually get interested and involved in contributing to the source code as well. 	Chapter 7
3.4. What are the challenges of crowdsourcing documentation?	<ul style="list-style-type: none"> - Maintaining the community effort requires significant input from a full-time community leader/champion. - Leading the project and community's work takes time. - Keeping the momentum is one of the key goals, but it requires a lot of work to do this. - Documentation production is not beneficial for scientists' reputation and careers, therefore there is little incentive for the user community to contribute time and effort. 	Chapter 7

Table 9.3: Study 3. Summary of findings mapped to the research questions

9.2 Further work

The richness of the data revealed a number of topics which, although they were not discussed in this dissertation as they did not fit into the main focus of the whole research, merit further study.

The outcomes of the preliminary study (chapter 2) indicated that it may be worth studying a wider range of scientific software commercialization cases and possibly proposing models for such commercialization processes. This could provide guidance for individuals and institutions who are considering commercializing a piece of scientific software. It may also be interesting to look into the cases of scientific software commercialization that did not succeed, and to

compare them to cases which turned out to be successful, in order to identify the factors that contribute to a success or a failure of scientific software commercialization. For example, it could be that, if the science behind the software is still evolving rapidly, such software may not be suitable for commercialization. The scientific model needs to be well understood and established before the software which it underpins can be commercialized. It is also possible that commercialization failures result from the mismatch between scientist-developers' expectations and ideas about the commercialization process and the reality of it. The scientists may underestimate the time and effort required to commercialize their software. They may not take into account a number of different actions which need to be undertaken in order to turn the software into a commercial product. A set of preliminary guidelines for commercializing software (as mentioned earlier) may also act as a set of criteria, a "check list" which could be used to gauge if a piece of software is ready to be commercialized.

Based on the outcomes of study 1 (chapter 5) it may be worthwhile to investigate further the transition from scientific end-user development to developing for a wider community of users. It may be interesting to look at the changes of practices other than documentation production related to software development. It may be that the changes occur on a very low level, for example, the coding style evolves together with the style and the contents of the comments. Alternatively, the priorities given to the required changes in the software may shift, and the scientist-developers' view of scientific research may become very different from the view they took in the scientific end-user context. The transition means a change in career, and a related discussion about the role of a "research software engineer" has been present in the popular (Parr, 2013) and academic discourse (Baxter et al., 2012). In the academic research context the evaluation of individuals is based on their publications rather than any other output which they produce. Scientist-developers who make a transition towards developing software for others are typically unable to publish as many papers as their peers, because most of their time and effort is allocated to software development. In effect, they do not receive relevant recognition and reward for their work, which leads many of them to leave academia to work in industry. There are many leads which the future research could undertake investigating in depth the situation of the "research software engineers", their main concerns, struggles and motivations. The

outcomes of such research could potentially be useful within academia, especially for those lobbying the funding bodies to create a career path for “research software engineers”.

The findings from both study 1 (chapter 5) and 2 (chapter 6) showed that the user community in scientific software represents a continuum from end-users (black-box users) to user-developers (white-box users). It may be worthwhile to investigate if this continuum occurs in other contexts of software development and what the consequences of such a phenomenon may be. Most user-developers initially develop and customize software for their own use. It would be interesting to investigate how often and in what circumstances they share their work. What may be the reasons for them to share? Are they, for example, influenced by the culture of their respective community to share the contributions to the source code? Another interesting aspect is related to the value of contributions to the project which user-developers make. In some situations the active engagement of user-developers is much needed and welcomed, as it increases the workforce available to progress the project, brings in new viewpoints and expertise. Possibly, the influence of user-developers may significantly change the direction of the project. However, it may also be that sometimes user-developers’ contributions bring more problems than benefits - the focus of the project may dissolve with a number of different viewpoints being thrown into it. The solution to that may be directing and actively using the user community potential. It may extend beyond just incorporating the source code which they produced. For example, professional software companies consult and engage their users (or simply customers, in case of commercial software) at different stages of software development. User innovation studies (for example, Von Hippel 2009) may provide some further ideas for developing this directions of research.

Another strand for further work could explore the topic of user-generated documentation in scientific software. Other case studies of scientific documentation crowdsourcing could bring interesting questions about similarities and differences with the insights from study 3 (chapter 7) reported in this dissertation. Exploration of other ways of capturing and sharing the users’ knowledge about the software (such as discussion fora or recently popular StackExchange¹) could be worth attention. It would be interesting to make comparisons to user-generated soft-

¹<http://stackexchange.com/>

ware documentation in the cases of other, non-scientific, software packages. What could be of particular interest is the question about the need for domain knowledge for software documentation. In scientific software, documenting software is closely tied up with documenting science which underpins it. In the cases in which software is not underlined by a scientific model, is it easier for any user to contribute to documentation?

References

- K. Ackroyd, S. Kinder, G. Mant, M. Miller, C. Ramsdale, and P. Stephenson. Scientific software development at a research facility. *IEEE Software*, 25(4):44–51, 2008. doi: 10.1109/MS.2008.93.
- A. Apon, S. Ahalt, V. Dantuluri, C. Gurdiev, M. Limayem, L. Ngo, and M. Stealey. High performance computing and research productivity in u.s. universities. *Journal of Information Technology Impact*, 10:87–98, 2010. URL <http://www.jiti.net/v10/jiti.v10n2.087-098.pdf>.
- J. Aranda, S. Easterbrook, and G. Wilson. Observations on conway’s law in scientific computing. In *1st Workshop on Socio-Technical Congruence (STC), at the 30th International Conference on Software Engineering (ICSE’08), 10 May 2008*, Leipzig, Germany, 2008. URL <http://www.cs.utoronto.ca/~jaranda/pubs/Aranda08-STC-final.pdf>.
- R. P. Bagozzi and U. M. Dholakia. Open source software user communities: A study of participation in linux user groups. *Management Science*, 52(7):1099 – 1115, 2006. ISSN 00251909. doi: 10.1287/mnsc.1060.0545. URL <http://mansci.journal.informs.org/content/52/7/1099.short>.
- V. R. Basili, D. Cruzes, J. C. Carver, L. M. Hochstein, J. K. Hollingsworth, M. V. Zelkowitz, and F. Shull. Understanding the high-performance-computing community: A software engineer’s perspective. *Software, IEEE*, 25(4):29–36, 2008. doi: 10.1109/SECSE.2009.5069158.
- R. Baxter, N. C. Hong, D. Gorissen, J. Hetherington, and I. Todorov. The research software engineer. In *Digital Research Conference, Oxford*, 2012. URL <http://dirkgorissen.com/2012/09/13/the-research-software-engineer/>.
- S. M. Baxter, S. W. Day, J. S. Fetrow, and S. J. Reisinger. Scientific software development is not an oxymoron. *PLoS Comput Biol*, 2(9):e87, 2006.

- A. Begel. End user programming for scientists: Modeling complex systems. In M. H. Burnett, G. Engels, B. A. Myers, and G. Rothermel, editors, *End-User Software Engineering*, number 07081 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
- E. Berglund and M. Priestley. Open-source documentation: in search of user-driven, just-in-time writing. In *Proceedings of the 19th annual international conference on Computer documentation*, SIGDOC '01, pages 132–141, New York, NY, USA, 2001. ACM. ISBN 1-58113-295-6. doi: 10.1145/501516.501543.
- H. Beyer and K. Holtzblatt. Contextual design. *Interactions*, 6(1):32–42, January 1999. ISSN 1072-5520. doi: 10.1145/291224.291229.
- R. E. Boyatzis. *Thematic analysis : coding as a process for transforming qualitative information*. Sage Publications, Thousand Oaks, CA, 1998.
- L. C. Briand. Software documentation: How much is enough? In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, CSMR '03, pages 13–15, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1902-4. doi: 10.1109/CSMR.2003.1192406.
- J. Brown and P. Duguid. *The social life of information*. Harvard Business Press, 2002.
- M. Burnett, C. Cook, and G. Rothermel. End-user software engineering. *Communications of the ACM*, 47:53–58, 2004.
- A. Calder, J. Dursi, B. Fryxell, T. Plewa, G. Weirs, T. Dupont, H. Robey, J. Kane, P. Drake, B. Remington, G. Dimonte, J. Hayes, J. Stone, P. Ricker, F. Timmes, M. Zingale, and K. Olson. Validating astrophysical simulation codes. *Computing in Science Engineering*, 6(5):10–20, Sept.-Oct. 2004. ISSN 1521-9615. doi: 10.1109/MCSE.2004.44.
- J. Carver. Empirical studies in end-user software engineering and viewing scientific programmers as end-users. In *End-User Software Engineering*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. URL <http://drops.dagstuhl.de/opus/volltexte/2007/1079/>.

- J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post. Software development environments for scientific and engineering software: A series of case studies. In *Proceedings of the 29th international conference on Software Engineering, ICSE '07*, pages 550–559, Washington, DC, USA, 2007. IEEE Computer Society. doi: 10.1109/ICSE.2007.77.
- P. K. Chilana, E. Fishman, E. M. Geraghty, P. Tarczy-Hornoch, F. M. Wolf, and N. R. Anderson. Characterizing data discovery and end-user computing needs in clinical translational science. *Journal of Organizational and End User Computing (JOEUC)*, 23(4, SI):17–30, Oct-Dec 2011. ISSN 1546-2234. doi: 10.4018/joeuc.2011100102.
- J. W. Creswell. *Research design : qualitative, quantitative, and mixed method approaches*. Sage, Thousand Oaks, Calif. ; London, 2003.
- B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, FSE '10*, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882312.
- S. Das, W. G. Lutters, and C. B. Seaman. Understanding documentation value in software maintenance. In *Proceedings of the 2007 Symposium on Computer human interaction for the management of information technology*, page 2. ACM, 2007.
- M. de Jong and L. Lentz. Professional writers and empathy: Exploring the barriers to anticipating reader problems. In *2007 IEEE International Professional Communication Conference*, pages 139–146. Seattle, WA, October 01-03, 2007, 2007. ISBN 978-1-4244-1242-6. doi: 10.1109/IPCC.2007.4464058.
- D. De Roure and C. Goble. Software design for empowering scientists. *Software, IEEE*, 26(1): 88–95, 2009. doi: 10.1109/MS.2009.22.
- S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information, SIGDOC '05*, pages 68–75, New York, NY, USA, 2005. ACM. ISBN 1-59593-175-9. doi: 10.1145/1085313.1085331.

- P. D'Este and M. Perkmann. Why do academics engage with industry? the entrepreneurial university and individual motivations. *Journal Of Technology Transfer*, 36(3):316–339, Jun 2011. ISSN 0892-9912. doi: 10.1007/s10961-010-9153-z.
- A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Communications of the ACM*, 54(4):86–96, Apr. 2011. ISSN 0001-0782. doi: 10.1145/1924421.1924442.
- P. F. Dubois. Maintaining correctness in scientific programs. *Computing in Science & Engineering*, 7(3):80–85, 2005. doi: 10.1109/MCSE.2005.54.
- S. Easterbrook and T. Johns. Engineering the software for understanding climate change. *Computing in Science & Engineering*, 11(6):65–74, 2009. ISSN 1521-9615. doi: 10.1109/MCSE.2009.193.
- S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. *Guide to advanced empirical software engineering*, pages 285–311, 2008. doi: 10.1007/978-1-84800-044-5_11.
- B. Einarsson, editor. *Accuracy and reliability in scientific computing*. Society for Industrial and Applied Mathematics, Philadelphia, 2005.
- S. Faulk, E. Loh, M. L. V. D. Vanter, S. Squires, and L. G. Votta. Scientific computing's productivity gridlock: How software engineering can help. *Computing in Science & Engineering*, 11(6):30–39, 2009. doi: 10.1109/MCSE.2009.205.
- J. Fereday and E. Muir-Cochrane. Demonstrating rigor using thematic analysis: A hybrid approach of inductive and deductive coding and theme development. *International Journal of Qualitative Methods*, 5(1):80, 2008. URL <http://ejournals.library.ualberta.ca/index.php/IJQM/article/download/4411/3530>.
- G. Fischer, E. Giaccardi, Y. Ye, A. Sutcliffe, and N. Mehandjiev. Meta-design: A manifesto for end-user development. *COMMUNICATIONS OF THE ACM*, 47(9):33–37, September 2004. ISSN 0001-0782. doi: 10.1145/1015864.1015884.
- S. Fomel and J. Claerbout. Guest editors' introduction: Reproducible research. *Computing in Science & Engineering*, 11(1):5–7, 2009. doi: 10.1109/MCSE.2009.14.

- S. R. Fussell and R. M. Krauss. Coordination of knowledge in communication: Effects of speakers' assumptions about what others know. *Journal of Personality and Social Psychology*, 62(3):378 – 391, 1992. ISSN 0022-3514. doi: 10.1037/0022-3514.62.3.378.
- R. L. Glass. Matching methodology to problem domain. *Communications of the ACM*, 47(5): 19–21, 2004. doi: 10.1145/986213.986228.
- M. A. Gurdon and K. J. Samsom. A longitudinal study of success and failure among scientist-started ventures. *Technovation*, 30(3):207–214, Mar 2010. ISSN 0166-4972. doi: 10.1016/j.technovation.2009.10.004.
- C. A. Halverson, C. Swart, J. Brezin, J. Richards, and C. Danis. Towards an ecologically valid study of programmer behavior for scientific computing. In *Proceedings of the First Workshop on Software Engineering for Computational Science and Engineering, ICSE*, volume 8, 2008. URL <http://secse08.cs.ua.edu/papers/halverson.pdf>.
- J. E. Hannay, H. P. Langtangen, C. MacLeod, D. Pfahl, J. Singer, and G. Wilson. How do scientists develop and use scientific software? In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 1–8, 2009. doi: 10.1109/SECSE.2009.5069155.
- J. Harrington. The SciPy documentation project. In G. Varoquaux, T. Vaught, and J. Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 33 – 35, Pasadena, CA USA, August 19-24 2008. URL http://conference.scipy.org/proceedings/scipy2008/paper_7/.
- J. Harrington and D. Goldsmith. Progress report: NumPy and SciPy documentation in 2009. In G. Varoquaux, T. Vaught, and J. Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 84–87, Pasadena, CA, August 18-23 2009. URL http://conference.scipy.org/proceedings/scipy2009/paper_14/full_text.pdf.
- L. Hatton and A. Roberts. How accurate is scientific software? *Software Engineering, IEEE Transactions on*, 20(10):785 –797, October 1994. ISSN 0098-5589. doi: 10.1109/32.328993.

- J. R. Hayes and D. Bajzek. Understanding and reducing the knowledge effect - implications for writers. *Written Communication*, 25(1):104–118, January 2008. ISSN 0741-0883. doi: 10.1177/0741088307311209.
- M. Heroux and J. Willenbring. Barely sufficient software engineering: 10 practices to improve your cse software. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 15–21, May 2009. doi: 10.1109/SECSE.2009.5069157.
- A. J. G. Hey and A. E. Trefethen. The uk e-science core programme and the grid. *Future Generation Computer Systems*, 18(8):1017–1031, September 2002. URL <http://eprints.soton.ac.uk/257644/>.
- C. Hine. Connective ethnography for the exploration of e-science. *Journal of Computer-Mediated Communication*, 12(2):618–634, 2007. ISSN 1083-6101. doi: 10.1111/j.1083-6101.2007.00341.x.
- K. Holtzblatt and S. Jones. Contextual inquiry: a participatory technique for system design. In D. Schuler and A. Namioka, editors, *Participatory design: principles and practices*, pages 177–210. Lawrence Erlbaum Associates, Hillsdale, 1993.
- D. Hook and D. Kelly. Testing for trustworthiness in scientific software. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 59–64. IEEE Computer Society, 2009. doi: 10.1109/SECSE.2009.5069163.
- S. E. Hove and B. Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Proceedings of the 11th IEEE International Software Metrics Symposium, METRICS '05*, pages 23–32, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2371-4. doi: 10.1109/METRICS.2005.24.
- J. Howison and J. D. Herbsleb. Scientific software production: incentives and collaboration. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work, CSCW '11*, pages 513–522, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0556-3. doi: 10.1145/1958824.1958904.

- D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(SI):W729–W732, July 1 2006. ISSN 0305-1048. doi: 10.1093/nar/gkl320.
- J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.55.
- S. Jain, G. George, and M. Maltarich. Academics or entrepreneurs? investigating role identity modification of university scientists involved in commercialization activity. *Research Policy*, 38(6):922 – 935, 2009. ISSN 0048-7333. doi: 10.1016/j.respol.2009.02.007.
- M. Jones and C. Scaffidi. Obstacles and opportunities with using visual and domain-specific languages in scientific programming. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 9 –16, September 2011. doi: 10.1109/VLHCC.2011.6070372.
- L. N. Joppa, G. McInerny, R. Harper, L. Salido, K. Takeda, K. O’Hara, D. Gavaghan, and S. Emmott. Troubling trends in scientific software use. *Science*, 340(6134):814–815, 2013. doi: 10.1126/science.1231535.
- M. Kajko-Mattsson. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering*, 10:31–55, 2005. ISSN 1382-3256. URL <http://dx.doi.org/10.1023/B:LIDA.0000048322.42751.ca>. 10.1023/B:LIDA.0000048322.42751.ca.
- D. Kelly. A software chasm: Software engineering and scientific computing. *Software, IEEE*, pages 119–120, Nov.-Dec. 2007. doi: 10.1109/MS.2007.155.
- D. Kelly and R. Sanders. Assessing the quality of scientific software. In *Proc of the First International Workshop on Software Engineering for Computational Science and Engineering*, 2008. URL <http://secse08.cs.ua.edu/Papers/Kelly.pdf>.
- D. Kelly, N. Cote, and T. Shepard. Software engineers and nuclear engineers: Teaming up to do testing. In *Proceedings of the Canadian Nuclear Society Conference, St. John, New Brunswick*, June 2007.

- D. Kelly, D. Hook, and R. Sanders. Five recommended practices for computational scientists who write software. *Computing in Science & Engineering*, 11(5):48–53, 2009. doi: 10.1109/MCSE.2009.139.
- D. Kelly, S. Smith, and N. Meng. Software engineering for scientists. *Computing in Science & Engineering*, 13:7, 2011a. doi: 10.1109/MCSE.2011.86.
- D. Kelly, S. Thorsteinson, and D. Hook. Scientific software testing: Analysis with four dimensions. *Software, IEEE*, 28(3):84–90, may-june 2011b. ISSN 0740-7459. doi: 10.1109/MS.2010.88.
- S. Killcoyne and J. Boyle. Managing chaos: Lessons learned developing software in the life sciences. *Computing in Science & Engineering*, 11(6):20–29, 2009. doi: 10.1109/MCSE.2009.198.
- B. A. Kitchenham, G. H. Travassos, A. von Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. Towards an ontology of software maintenance. *Journal of Software Maintenance*, 11(6):365–389, 1999.
- H. K. Klein and M. D. Myers. A set of principles for conducting and evaluating interpretive field studies in information systems. *MIS Quarterly*, 23(1):67(2), 1999. URL <http://www.jstor.org/stable/249410>.
- A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43:21:1–21:44, April 2011. ISSN 0360-0300. doi: <http://doi.acm.org/10.1145/1922649.1922658>.
- K. R. Lakhani and E. von Hippel. How open source software works: “free” user-to-user assistance. *Research Policy*, 32(6):923–943, 2003. ISSN 0048-7333. doi: 10.1016/S0048-7333(02)00095-1.
- A. Lam. What motivates academic scientists to engage in research commercialization: ‘gold’, ‘ribbon’ or ‘puzzle’? *Research Policy*, 40(10):1354–1368, DEC 2011. ISSN 0048-7333. doi: 10.1016/j.respol.2011.09.002.

- C. R. Lanier. Open source software peer-to-peer forums and culture: A preliminary investigation of global participation in user assistance. *Journal of Technical Writing & Communication*, 41(4):347 – 366, 2011. ISSN 00472816. doi: 10.2190/TW.41.4.c.
- T. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: the state of the practice. *Software, IEEE*, 20(6):35 – 39, Nov.-Dec. 2003. ISSN 0740-7459. doi: 10.1109/MS.2003.1241364.
- C. Letondal. Participatory programming: Developing programmable bioinformatics tools for end-users. *End User Development*, pages 207–242, 2006. doi: 10.1007/1-4020-5386-X_10.
- R. Leveque, I. Mitchell, and V. Stodden. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Computing in Science Engineering*, 14(4):13–17, 2012. ISSN 1521-9615. doi: 10.1109/MCSE.2012.38.
- M. Marshall. Sampling for qualitative research. *Family Practice*, 13(6):522–525, DEC 1996. ISSN 0263-2136. doi: 10.1093/fampra/13.6.522.
- Z. Merali. Computational science: Error, why scientific programming does not compute. *Nature*, 467(7317):775–777, 2010. doi: 10.1038/467775a.
- C. Morris. Some lessons learned reviewing scientific code. In *Software Engineering for Computational Science and Engineering, 2008. SECSE '08. ICSE Workshop on*, 2008. URL <http://secse08.cs.ua.edu/Papers/Morris.pdf>.
- B. Nardi. *A small matter of programming: perspectives on end user computing*. The MIT Press, Cambridge, 1993.
- L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana. A survey of scientific software development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10. ACM, 2010. doi: 10.1145/1852786.1852802.
- R. S. Nickerson. How we know and sometimes misjudge what others know: Imputing one’s own knowledge to others. *Psychological Bulletin*, 125(6):737 – 759, 1999. ISSN 0033-2909. doi: 10.1037/0033-2909.125.6.737.

- F. Niessink and H. Van Vliet. Software maintenance from a service perspective. *Journal of Software Maintenance*, 12(2):103–120, 2000.
- D. O’Leary. Computational software: writing your legacy. *Computing in Science Engineering*, 8(1):78 – 80, January–February 2006. ISSN 1521-9615. doi: 10.1109/MCSE.2006.3.
- C. Parnin, C. Treude, L. Grammel, and M. Storey. Crowd documentation: Exploring the coverage and the dynamics of API discussions on stack overflow. Technical report, Georgia Tech, 2005. URL http://larsgrammel.de/publications/parnin_2012_crowd_documentation.pdf.
- C. Parr. Save your work give software engineers a career track. Times Higher Education, 15 August 2013. URL <http://www.timeshighereducation.co.uk/news/save-your-work-give-software-engineers-a-career-track/2006431.article>.
- A. Pawlik, J. Segal, H. Sharp, and M. Petre. “Everything” about scientific software documentation that wasnt in the manual. In *Scientific Software Days 2012, 17th December 2012, Austin, TX, USA*, 17th December 2012a. URL <http://scisoftdays.org/meetings/2012/>.
- A. Pawlik, J. Segal, H. Sharp, and M. Petre. Documentation practices in scientific software development. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2012 5th International Workshop on*, pages 113–119, 2012b. doi: 10.1109/CHASE.2012.6223004.
- A. Pawlik, J. Segal, H. Sharp, and M. Petre. Developing scientific software: The role of the internet. In *Science and the Internet, International Conference on*, volume Dsseldorf, Germany, 1- 3 August 2012c. URL <http://nfgwin.uni-duesseldorf.de/sites/default/files/Pawlik.pdf>.
- A. Prlic and J. B. Procter. Ten simple rules for the open development of scientific software. *PLoS Comput Biol*, 8(12):e1002802, 12 2012. doi: 10.1371/journal.pcbi.1002802.
- M. Raven and A. Flanders. Using contextual inquiry to learn about your audiences. *ACM SIGDOC Asterisk Journal of Computer Documentation*, 20(1):1–13, 1996. doi: 10.1145/227614.227615.

- E. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999. ISSN 1874-6314. doi: 10.1007/s12130-999-1026-0.
- M. Rettig. Nobody reads documentation. *Commun. ACM*, 34(7):19–24, July 1991. ISSN 0001-0782. doi: 10.1145/105783.105788.
- H. Robinson, J. Segal, and H. Sharp. Ethnographically-informed empirical studies of software practice. *Information and Software Technology*, 49(6):540–551, 2007. doi: 10.1016/j.infsof.2007.02.007.
- C. Robson. *Real world research : a resource for social scientists and practitioner-researchers*. Blackwell, Oxford, 2002.
- E. M. Rogers. *Diffusion of innovations*. Glencoe: Free Press, 1962.
- P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley, 2012.
- R. Sanders and D. Kelly. Dealing with risk in scientific software development. *Software, IEEE*, pages 21–28, 2008. doi: 10.1109/MS.2008.84.
- C. J. Savage and A. J. Vickers. Empirical study of data sharing by authors publishing in plos journals. *PLoS ONE*, 4(9):e7078, 09 2009. doi: 10.1371/journal.pone.0007078.
- C. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999. doi: 10.1109/32.799955.
- C. B. Seaman. The information gathering strategies of software maintainers. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 141–149. IEEE, 2002.
- C. B. Seaman. Qualitative methods. In F. Shull, J. Singer, and D. I. K. Sjberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 35–62. Springer London, 2008. ISBN 978-1-84800-044-5. doi: 10.1007/978-1-84800-044-5_2.
- J. Segal. Two principles of end-user software engineering research. *SIGSOFT Software Engineering Notes*, 30:1–5, 2005a. ISSN 0163-5948. doi: 10.1145/1082983.1083240.
- J. Segal. When software engineers met research scientists: A case study. *Empirical Software Engineering*, 10:517–536, 2005b. ISSN 1382-3256. doi: 10.1007/s10664-005-3865-y.

- J. Segal. Some problems of professional end user developers. In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*, pages 111–118. IEEE, 2007.
- J. Segal. Models of scientific software development. In *SECSE 08, First International Workshop on Software Engineering in Computational Science and Engineering*, Leipzig, Germany, 13 May 2008a. URL <http://www.cs.ua.edu/~SECSE08/Papers/Segal.pdf>.
- J. Segal. Scientists and software engineers: a tale of two cultures. In *Proceedings: 20th annual meeting of the Psychology of Programming Interest Group; Lancaster, United Kingdom*, University of Lancaster, UK, September 10-12 2008b. ISBN 978-1-86220-215-3. URL <http://oro.open.ac.uk/id/eprint/17671>.
- J. Segal. Software development cultures and cooperation problems: A field study of the early stages of development of software for a scientific community. *Computer Supported Cooperative Work (CSCW)*, 18(5-6):581–606, 2009. ISSN 1573-7551. doi: 10.1007/s10606-009-9096-9.
- J. Segal and C. Morris. Developing software for a scientific community: some challenges and solutions. In J. Leng and W. Sharrock, editors, *Handbook of Research on Computational Science and Engineering: Theory and Practice.*, pages 177–196. USA: IGI Global, 2011. doi: 10.4018/978-1-61350-116-0.ch008.
- J. Singer. Practices of software maintenance. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 139–145. IEEE, 1998.
- I. Sommerville. *Software engineering*. Pearson / Addison-Wesley, Harlow, 2004.
- M. J. C. Sousa and H. M. Moreira. A survey on the software maintenance process. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 265–274. IEEE, 1998.
- S. Squires, M. L. Van De Vanter, and L. G. Votta. Yes, there is an “Expertise Gap” in HPC applications development. In *Third Workshop on Productivity and Performance in High-End Computing (PPHEC-06)*, IEEE CS Press, Austin, TX, 12 February 2006. URL http://research.sun.com/pls/apex/f?p=labs:40150:0::::P40000_PUBLICATION_ID:3522.

- V. Stodden. The legal framework for reproducible scientific research: Licensing and copyright. *Computing in Science Engineering*, 11(1):35–40, 2009. ISSN 1521-9615. doi: 10.1109/MCSE.2009.19.
- S. Thew, A. Sutcliffe, R. Procter, O. de Bruijn, J. McNaught, C. C. Venters, and I. Buchan. Requirements engineering for e-science: Experiences in epidemiology. *Software, IEEE*, 26(1):80–87, 2009. ISSN 0740-7459. doi: 10.1109/MS.2009.19.
- S. J. Van der Walt. The SciPy documentation project (technical overview). In G. Varoquaux, T. Vaught, and J. Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 27 – 28, Pasadena, CA USA, 2008. URL <http://www.cs.ua.edu/~SECSE08/Papers/Segal.pdf>.
- P. Vandewalle, J. Kovacevic, and M. Vetterli. Reproducible research in signal processing - what, why, and how. *IEEE Signal Processing Magazine*, 26(3):37–47, May 2009. ISSN 1053-5888. doi: 10.1109/MSP.2009.932122.
- M. Vidger, N. G. Vinson, J. A. Singer, D. Stewart, and K. Mews. Supporting the everyday work of scientists: Automating scientific workflows. *Software, IEEE*, 25(4):52–58, 2008. doi: 10.1109/MS.2008.97.
- E. Von Hippel. Democratizing innovation: the evolving phenomenon of user innovation. *International Journal of Innovation Science*, 1(1):29–40, 2009. ISSN 1757-2223. doi: 10.1260/175722209787951224.
- E. Wenger. *Communities of practice: Learning, meanings, and identity*. New York: Cambridge University Press., 2007.
- E. Wenger, R. McDermott, and W. Snyder. *Cultivating communities of practice: A guide to managing knowledge*. Boston, Mass: Harvard Business School Press., 2002.
- G. Wilson. Software carpentry: Getting scientists to write better code by making them more productive. *Computing in Science & Engineering*, 8(6):66–69, 2006. ISSN 1521-9615. doi: 10.1109/MCSE.2006.122.
- G. Wilson. Those who will not learn from history. *Computing in Science and Engineering*, 10(3):5, 2008.

W. Wood and W. Kleb. Exploring XP for scientific research. *Software, IEEE*, 20(3):30 – 36, May–June 2003. ISSN 0740-7459. doi: 10.1109/MS.2003.1196317.

R. K. Yin. *Case study research : design and methods*. Sage, Los Angeles, California ; London, 2009.

Appendix 1. Informed consent form



The Open University
Computing Department
Walton Hall
Milton Keynes, MK7 6AA
United Kingdom

Research Team
Aleksandra Pawlik, (PhD student)
Dr. Judith Segal, Prof. Marian Petre, Prof. Helen Sharp (Supervisors)

INFORMED CONSENT FORM

I, _____ agree to participate in this study to investigate the experiences of scientists who develop a variety of scientific software. I understand that this study is a part of Aleksandra Pawlik's PhD research at the Open University (working title "**Effective Support for the Scientific Software Development Practices of Computational Scientists**").

I agree to be interviewed and acknowledge that the interview will be recorded and then transcribed.

I understand that all data collected during the study is confidential, and that the data will be anonymised for analysis and use. The original recordings and any other documents collected from me during the study will be stored in a secure manner, and can be accessed only by the Research Team.

I understand that I can withdraw from the study at any time and that I can request that my data be removed

I understand that I will be invited to review any paper in which my data is used before submission for publication.

Signed

Date

Appendix 2. Preliminary study - interview questions

Introductory questions

1. What is your background in the scientific software development?
2. What is the story of commercializing your software?

Scientific software commercialization

1. What were the reasons for transforming your software prototype to a releasable product?
2. What were your experiences during the whole process?
3. What were your main goals and aims?
4. What were the main difficulties you encountered?
5. What worked well during that process?
6. Was there anything that surprised you?
7. Can you describe your interaction with other developers involved in the process?
8. Can you describe your interaction with the end-users (anyone who used the software at its different stages)?

Appendix 3. Study 1 - interview questions

Introductory questions

1. How did you move on from developing scientific software mainly for your own purposes to the development for a wider community of users?
2. What kind of scientific software you develop currently?
3. Have you ever worked as a software developer outside purely research/academic environment, in the industry/business?

Documentation practices:

1. Can you describe how does a typical handover of a copy of your software to a user look like?
 - What do you provide in the software package (what kind of materials, information etc.)?
 - What are the forms of the elements in this package?
2. Can you compare the ways of capturing your software development practices now and before, when you developed mainly for your own purposes?
3. How do you communicate (about things related to software development) within the software development team?
 - Is there any formally established way (a protocol, a procedure) of handling this communication?
 - What are the forms of this communication?
 - What is the frequency of this communication?
 - Have you ever introduced a new scientists (as a developer) to the project? If yes, can you describe how you did that?
4. How do you communicate with the users?
 - Who are the users that you communicate about the software?
 - What do you discuss in this communication?
 - How frequent is this communication?
 - How do you capture what is discussed with the users?
 - How do your process further this captured information?
 - How is this information used by other members of the development team?
5. How do you capture information describing the design aspect of the software?
 - How do you design the software (a software module)?
 - How do you capture information about the software architecture ?
 - What is the form of capturing information about the software architecture?
 - When is this kind of information captured (after/during the meetings etc.)?
 - Who uses this kind of captured information?
6. How do you capture the technical aspect of the software?
 - How do you capture the information about the source code itself (comments within the code, code structuring)?
 - Is there any standard, any procedure which for capturing the information about the technical aspects of the code?
 - Who uses this kind of captured information?
7. How do you acquire knowledge about capturing software development activities?
 - Do you have any formal training in software development /software engineering (a degree, a course etc.)

If yes, please provide more details about it.

If no, are you self-taught? What are the resources that you use to learn software development?

- What are your skills related do capturing software development activities

8. Do you prepare user manuals?

- How is user manual created?
- What is the form of this manual?
- Do you have any procedure (a protocol) of developing user manuals?

9. Can you notice any problems with capturing the software development activities? If yes, can you discuss this in more detail?

Appendix 4. Study 2 - interview questions

Introductory questions:

1. What is your experience in using scientific software?
2. What kinds of scientific software packages have you used (for example, it was a Windows application with GUI and now they use a batch-type software run or UNIX)?
3. Do you have any experience in scientific software development?
4. If you have such experience, how do you interact with the end-users?

Work-based interview questions prompt list:

1. In what situations do you look for resources which help you to use the software?
2. What are the resources of information and documentation do you use when working with this software?
3. How often do you look for these resources?
4. How do you find these resources/the information about the software?
5. Why do you use these methods of searching for the information about the software?
6. What are the forms of these resources/the documentation related to the software that you use?
7. What are the advantages/disadvantages of these forms?
 - Would you change them?
 - Why?
8. How about consulting your colleagues (if the participant mentions only written/drawn materials), do you consult them about the software?
9. How do you know that the resources which you use provide the correct information/the information that you are actually looking for?
10. How would you rate the importance of these resources / documentation for you?
11. Are there any issues related to these resources/documentation?
 - What are these issues?
 - Why do they occur?
 - How do you address them?
12. What other resources/documentation should be made available?
13. How much time does it take you (on average, if you can estimate) to get the information you need about the software?
14. Were there any situations in which it took you a very long time to find the information about the software which you needed?
15. If you could get this information faster, what could you spend this time on?
16. Can you think of any, other than time, losses resulting from the fact that the information you needed was not available/difficult to access?

List of proposed activities to be recreated / recalled (during the artefact walkthrough):

1. software installation;
2. a problematic situation:
 - a runtime error (a bug in the software);
 - problems with understanding the scientific functionality of the software (eg. having difficulties with setting up a particular simulation);
 - problems with understanding technical requirements of the software (eg. what data format is acceptable);
3. adding a new functionality (in case of users-developers) eg. experiencing problems with understanding technical specification and already implemented solutions.

Appendix 5. Study 3 - interview questions

Introductory questions

1. What is your background in scientific software development?
2. How did you get involved in the SciPy/NumPy community?
3. What is/was your role in the community?
4. What is/was your role in the documentation project?

The history of the SciPy and NumPy documentation project

1. Can you recall how documentation crowd sourcing started?
2. Do you remember how crowd sourcing documentation was implemented?
3. Why did you decide to do it?
4. When you are thinking about the process now what would you change about it?
5. Do you remember any issues when you were implementing the process?
6. What standards and formats of documentation were used and why?
7. What is/was the Accessible SciPy (ASP) project?
8. What were the Documentation Marathons?

Current work on the documentation for SciPy and NumPy

1. Can you describe the process of documentation production in SciPy and NumPy as it is now?
2. Can you tell me more about the documentation workflow depicted on the the documentation website?
3. What would you like to change anything about the process?
4. Are there any issues? If yes, what kind of issues?
5. Who is involved in producing documentation?
6. Is there a core team working on documentation? Is there a leader of the project? Who is it?
7. How does the leader communicate with the community (in particular, with those who contribute to documentation)?
8. How many people are allowed to edit documentation? Do you keep any kind of a list? Is this list revised?
9. What is the verification process of the people who want to edit documentation online (what happens between someone registering and asking for permission and the moment of being given the permission)?
10. Are people who contribute to documentation regularly active? Can you see any patterns in contributions made to documentation?
11. Who are the people producing documentation? (their background, degree, current job etc.) Do you know why they want to produce documentation?
12. How often new version of documentation is made available? Are there any release cycles?
13. How do you inform the community about new documentation?

Source code development in SciPy and NumPy. Additional questions aiming to elicit more information / gain more context

1. Can you describe the process of source code development in SciPy and NumPy?
2. Who can contribute to the source code?
3. How did you arrive to this process?
4. Do you remember how it was implemented?
5. Would you change anything about this process?
6. What are, in your opinion, the advantages of this process of source code development? What are the disadvantages?

7. How many developers have write access to the repository?
8. Is the list of those who have access revised on the regular basis? Who revises this list?
9. Are there any dormant developers?
10. What happens to their credentials? Do credentials expire after some time if the account is not used?
11. Who revises the list of people who have write access to the repository?
12. What kind of an existing developer can recommend a developer to have a write access?
13. How does this number of developers who have access to the repository change? Is it steadily increasing?
14. Who are the people contributing to the source code (their background, degree, current job etc.)? Do you know why they want to contribute to the source code?
15. How often is new code released? How do you inform the community about it?
16. Is there a core development team? Is there a leader of the project/development? Who is it?
17. How does the leader communicate with the community who develops the code?

Appendix 6. Coding schemes

Coding scheme for the preliminary study - software commercialization

Academic - Commercial	
Name	
	Academic-Environment-Perception
	Academic-Phase-Code-Sharing
	Commerical-Funding-for-Research
	Difference-Commercial-Science-Academic
	Software-Academic-Phase-Characteristics
	Software-Academic-Phase-Sharing
	Software-Commercialization-Research
	Software-Development-Combining-Academic-Commercial

Challenges	
Name	
	Software-Academic-Phase-Obstacles
	Software-Commercialization-Problems-Marketing
	Software-Development-Now-Problems
	Software-Development-Past-Difficulties

Commercialization process and actions	
Name	
	Intellectual-Property-University-Software-Profit
	Software-Commercialiation-IP-Licensing
	Software-Commercialisation-Start-Reasons
	Software-Commercialization-Copyrights
	Software-Commercialization-Creating-Company
	Software-Commercialization-Finding-Customers
	Software-Commercialization-Funding
	Software-Commercialization-Relationship-Power
	Software-Commercial-Phase-Company-Development
	Software-Start-Funding

Evolution of Software	
Name	
	Software-Commercial-Phase-Changes
	Software-Development-Academic-Phase-Improvements-Usage
	Software-Development-Anticipation-Technology
	Software-Development-Anticipations
	Software-Development-Milestone-Changes
	Software-Development-Milestones-Technology
	Software-Development-Stages
	Software-Performance-Technology

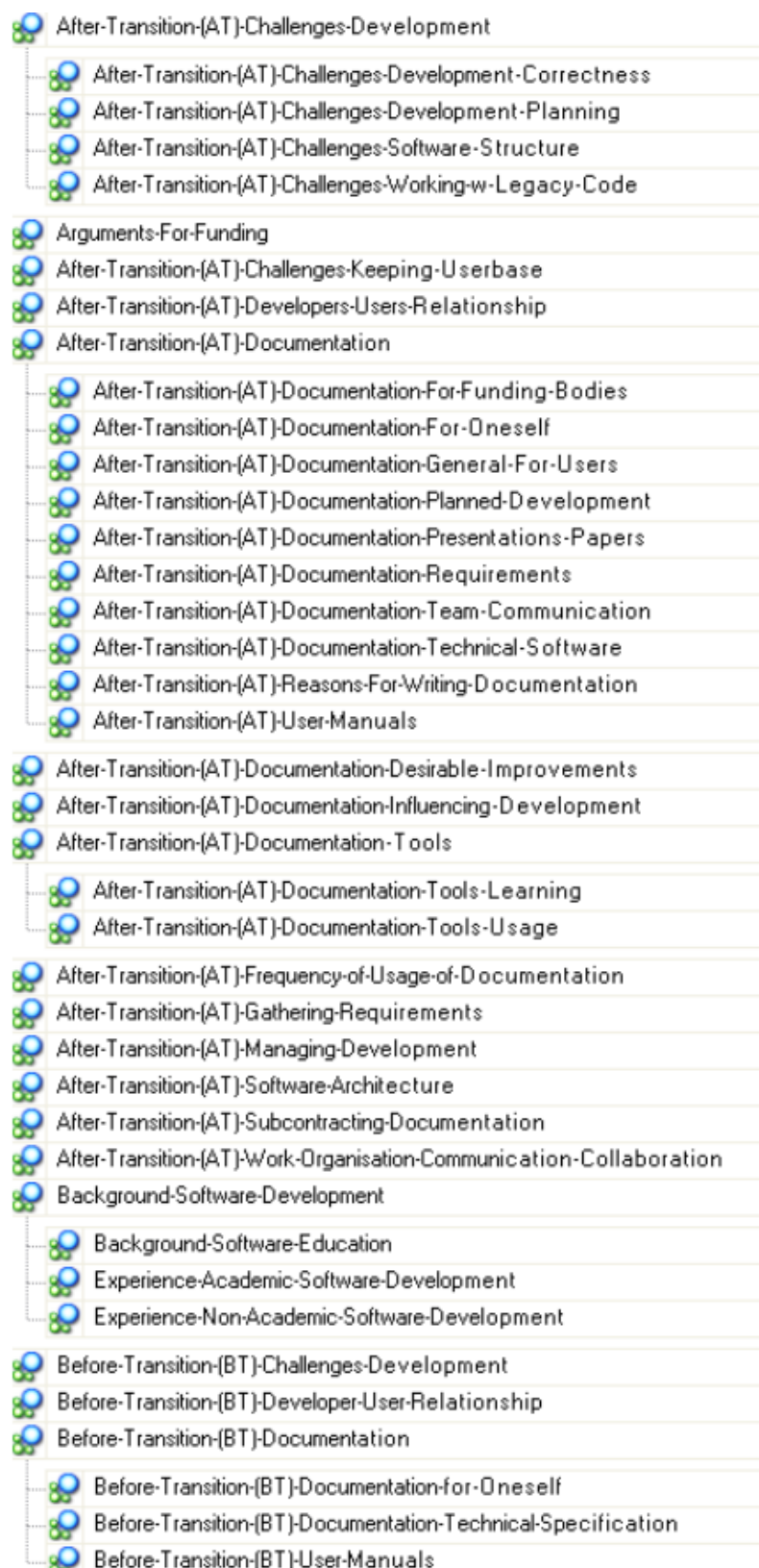
Future of Software	
Name	
	Software-Development-Future-Problems
	Software-Development-Parallel-Architectures
	Software-Development-Plans-Future

Science and Sci. SW Perception	
Name	
	Attitude-Towards-Science
	Scientific-Software-Perception

Software Developers	
Name	
	Carreer-Change-Towards-Software-Development
	Self-perception
	Software-Commercialization-Change-Team
	Software-Commercialization-Finding-Partners
	Software-Development-Collaboration
	Software-Development-People-Background
	Software-Start-Graduate Students
	Software-Start-People

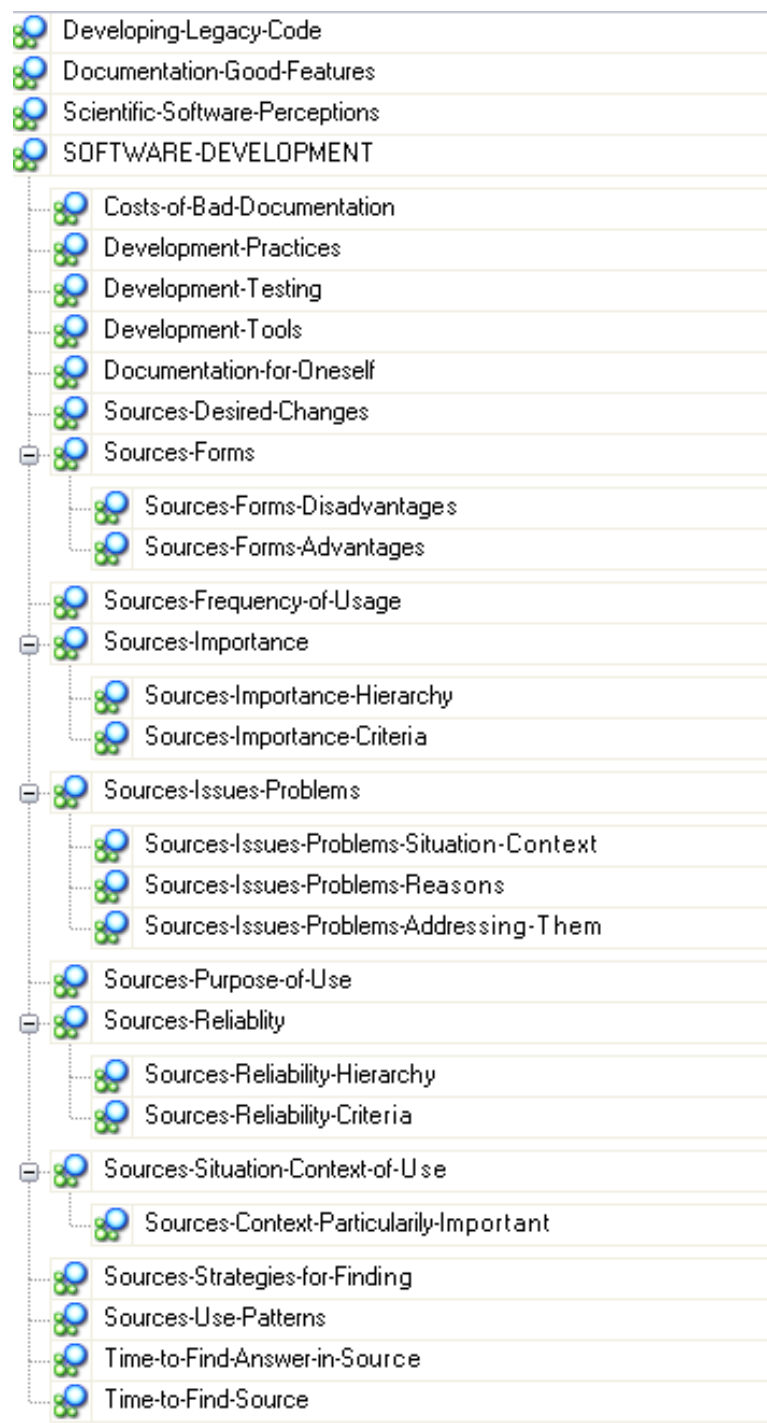
Software Development Practice	
Name	
Software-Versions-Mainframes	
Software-Support-Maintenance	
Software-Start-Work-Organisation	
Software-Documentation-Manual	
Software-Documentation-Design-Models	
Software-Development-Work-Organisation	
Software-Development-Workload	
Software-Development-Windows-OS	
Software-Development-Testing	
Software-Development-Source-Code-Structure	
Software-Development-Programming Languages	
Software-Development-Operating-Systems	
Software-Development-Now-Code-Ownership	
Software-Development-Linux-Version	
Software-Development-Learning	
Software-Development-Implementing-Science	
Software-Development-IDE	
Software-Development-Fortran	
Software-Development-Factors-Technology	
Software-Development-Documentation	
Software-Development-Debugging-Testing	
Software-Development-Data Structures	
Software-Development-Cross-Platform	
Software-Development-Commercial-Phase-Programming	
Software-Development-Code-Ownership	
Software-Development-C	
Software-Commercialization-Beginning-Work-Organisation	
Object-Oriented-Usage	
Object-Oriented-Programming-Perception	
Software Marketing	
Name	
Software-Exclusivity-to-Customers-Marketing	
Business-Marketing-Work-Organisation	
Software users	
Name	
Software-Start-Initiative	
Software-Development-Users-Role	
Software-Development-Initiatives	
Software-Commercial-Customer-Groups	
Software users	
Name	
Software-Development-Easy	
Software-Development-Strengths	
Software-Engineering-Perception	
Software users	
Name	
Changes-Perception-New Programming Languages	
Software users	
Name	
Software-Start-Technology-Factors	

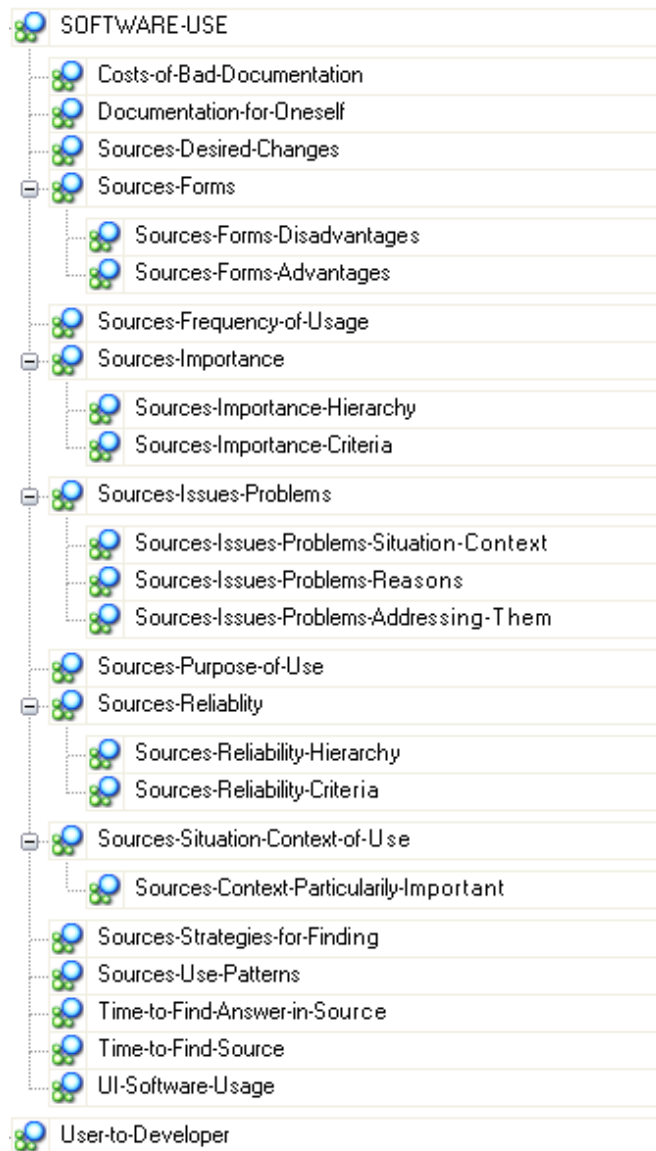
Coding scheme for study 1 - scientific software documentation production

































- Before-Transition-(BT)-Planning-Development
- Collaboration-Scientists-Software-Engineers
- Combining-Scientist-Developer-Roles
- Differences
 - Differences-Development-Before-Transition-(BT)-vs-After-Transition-(AT)
 - Differences-Documentation-Before-Transition-(BT)-vs-After-Transition-(AT)
 - Differences-Scientific-Industrial-Development
- General-Comments
 - General-Approach-to-Development-by-Scientists
 - General-Comments-Documentation-Scientific-Software
 - General-Comments-Issues-Scientific-Software
- Inspiration-Documentation-Changes
- Inspirations-for-Development-Management-Changes
- Learning-about-Documentation
- Problems-Documentation
 - After-Transition-(AT)-Problems-Documentation
 - After-Transition-(AT)-Problems-Technical-Documentation
 - After-Transition-(AT)-Problems-Documentation-Team-Communication
 - After-Transition-(AT)-Problems-Documentation-for-Users
 - After-Transition-(AT)-Problems-Documentation-for-oneself
 - Before-Transition-(BT)-Problems-Documentation
- Problems-Recognition-Scientific-Soft-Development
- Propagating-Software-Practices
- Sustainability-Curation-of-Scientific-Software
- User-Knowledge-Assumptions
- User-Types
 - Before-Transition-(BT)-User-Types
 - User-Types-After-Transition-(AT)
 - User-Types-After-Transition-(AT)-End-User-Developers
 - User-Types-After-Transition-(AT)-End-Users



































Coding scheme for study 2 - scientific software documentation use





Coding scheme for study 3 - scientific software documentation crowdsourcing

	Agruments-For-Wiki
	Arguments-for-Crowd-Sourcing
	Arguments-For-Livedocs
	Arguments-For-PDF-Documentation
	Accessible-SciPy-Project-(ASP)-Documentation
	Accessible-SciPy-Project-(ASP)
	Choosing-Documentation-Tools-by Users
	Conflict-Documentation-Style-Convention
	Docbooks
	Documentation-Book
	Documentation-Engaging-Users
	Documentation-Formats-Tools-Poll
	Documentation-Forms
	Documentation-Pattern-of-Use
	Documentation-Quick-Start-Tutorial
	Documentation-Standards-Queries
	Documentation-Tools
	Documenting-Formulae
	Existing-Documentation
	Existing-Documentation-Conversion
	Finding-Developers-for-Code
	Implementation
	Module-Submission-Review-Process
	Implementation-Documentation-Website
	iPython-documentation
	iPython-to-include-in-SciPy
	Issues
	Doc-Crowd-Source-Implementation-Issues
	Issues-w-Documentation
	Issues-w-Tools-Documentation

 LaTeX-ReST-Word-Tables
 LyX-advantage
 Motivating-Community
 Project Organisation Suggestions
 Reasons-for-not-Producing-Documantation
 SciPy-Journal
 SciPy-ScientificPython
 Spontaneously-User-Contributed-Documantation
 Suggestion-Documantation-Examples
 Suggestion-Documantation-Feedback-Importance
 Suggestion-Documantation-Live-Documantation
 Suggestion-Documantation-Offline
 Suggestion-Documantation-Standard-Importing
 Suggestion-Documantation-Wiki
 Suggestion-New-Users-Engagement
 Suggestion-Process-Contributing-Code
 Suggestions
 Developer-Doc-Implementation-Suggestion
 Suggestion-Docstrings
 Suggestion-Documantation-Contents
 Suggestion-Modules-Documantation
 Suggestions-Documantation
 Suggestions-Tutorials-Instructions
 Technical-Writer-Doc-Implementation-Suggestion
 User-Suggestion-about-Doc-Implementation
 User-Suggestions-Complaints-on-Documantation
 User-Suggestion-to-Contribute-to-Doc
 Suggestions-Documantation-on-Website
 Suggestions-Organising-Process
 Suggestions-Tools
 Tools-RST-ST-Word-LaTeX
 Types-of-Documantation
 Users-Contributions-to-Documantation
 Ways-to-Engaging-Community

Appendix 7. Data - excerpts from the interviews and mailing list archive

This Appendix includes excerpts from the interviews and the mailing list archive (for Study 3) which correspond with the presentation of the findings in the empirical chapters of this thesis (Chapters: 2, 5, 6 and 7).

Chapter 2

Quote 2-1

At some point, within 2-3 weeks, we received two inquiries [about the software]. One came from Japan and another one from France, from two organisations. The one from Japan was from company X [name removed] and the one from France was from a nuclear lab. They asked “Could we purchase your application?”. [It was] because people somehow, somewhere through their professional research contacts came across this tool that was potentially useful for them, for their work. And so they send their inquiry because they wanted to do it properly and have a legal copy. (...) And then the idea occurred that if someone wants to buy it, it should be somehow formalized [Interviewee A]

Quote 2-2

I think it's grown more or less as we went along. I don't think we started with a plan in which we take in a number of components and the number of properties that we're handling now. That's causing us a certain amount of headaches. I think we could have offered some new properties on a shorter time scale, if we had a better architecture of the thing from the beginning. [Interviewee E]

Quote 2-3

We weren't really spending time on the documentation. In the end it wasted a lot of time because we couldn't remember quite what we did. The programs are getting very long now because we've got all these add-ons so I think one of our party here said “We must from now on be much tidier with our paperwork”. [Interviewee E]

Quote 2-4

You don't document the source program sufficiently, don't make it general enough so that you could make changes in the future. When you do the recoding, “the spaghetti” where you go back to the code you wrote 10 years ago and you can't remember how it works so you can't modify it. [Interviewee D]

Quote 2-5

I have never done any OO programming before. I realized that if I was going to program in Java that I had to [practice]. So I wrote a few kind of “learning how to do it” programs. (...) I just kind of learnt as I went along. But Java, I have to say is a very easy language to learn. If I compare that with my experience when I was using FORTRAN, that was you know...30+ years ago as an undergraduate and that was for me very hard to learn, it wasn't obvious. It wasn't easy just to program to run. If I contrast, then I'd say learning Java was really pretty

easy. [Interviewee G]

Quote 2-6

I must say that this is truly changing the complexity of the software and the complexity of the models that we start to use. It needs the person who does the modeling or somebody who understands modeling fairly well, that needs to do the programming. [Interviewee F]

Quote 2-7

we don't see ourselves here as software engineers, we see ourselves primarily as university professors doing research, organized around the commercial product - it's a part of it. Most of what I like doing and what I do is research. [Interviewee D]

Chapter 5

Quote 5-1

I reckon I repeated 3.5 years of work in 6 months at the end of my PhD. If stuff had been better documented, then it would have probably been more like 2 months. I probably wasted 4 months retrying the wrong thing because I had not made sufficiently good notes as to what I had done. [Interviewee U]

Quote 5-2

when I started modifying code in the UK I used to put comments in Spanish too because it was the thing that came to my mind and you realize there is going to be other people looking at this, you can't do this any more. You've got to try and do it in a way that people can follow. [Interviewee L]

Quote 5-3

I actually have a thing called a user manual. But I do separate some of the details. Some of the programming detail goes into the appendices rather than the main body of documentation. The main documentation will contain the details that all users will need to be able to run the code and model the particular systems. [Interviewee P]

Quote 5-4

the codes I'm working on now we're kind of more moving towards more intelligent documentation so documentation within the source code of the API, using wikis and that sort of stuff. So that the users can actually help us keep the documentation up. [Interviewee D]

Quote 5-5

Nothing formal. We have regular meetings - the developers that I work with [are] actually here on site, so we can see each other face-to-face and...maybe in the meeting you may put a slide up with a list of things you think should be done but there is no formal shared area where you put these things. The problem with shared things, wikis and stuff like that is that this kind of environment hasn't really caught on here. Maybe there aren't enough young people around that share information through a wiki or something. Although it has caught on with people who worked with developers which are on other sites. But if you are at the same site you think it's better to talk face-to-face. If we created a wiki nobody would use it really. It would not be maintained so we tend to talk just face to face really. [Interviewee N]

Quote 5-6

[Q] How did you arrive at writing your documentation that way?

[Participant] It was more or less how it was when I received it. There was a user manual that came with the version of the software which existed. But there was at least one chapter which was "To be written...". But there was definitely a structure. I have slightly modified the structure. [Interviewee P]

Quote 5-7

It's a bit like teaching. When you start teaching you actually realize how much you don't understand about the subject you are teaching. (...)You start thinking "Why did I do it this way?". [Interviewee M]

Quote 5-8

A lot of the users use the code as black-box to a large extent. They don't want to and they would say they don't need to but I'd probably say they should, understand what's going in the code. It's still a very interesting code because we do expect the users to understand what's going on to get the good performance out of it. [Interviewee M]

Quote 5-9

We got the impression that most people didn't read it [the manual]. Most people would ask you how to do this rather than read it. But occasionally somebody would say "This isn't in the manual. I tried to do this with the manual and I couldn't find it". But I think for most people how detailed the documentation is wasn't really anything that they were interested in. [Interviewee O]

Chapter 6**Quote 6-1**

The point of going through that list was to learn about those I didn't know about. So then I go through the link there. I tend to pick sites I trust. I thought that was a good site. [Interviewee B]

Quote 6-2

It was mainly with my colleague. He started off using it. He knew how to use it before (...)He showed me how to use it. He explained how to use it to me personally. He is a very good teacher. It's very easy to follow him. But there are a number of other things you have to do yourself. That's the key with anything that has to do with computing. Somebody can run though it with you on the screen but you really have to try and do it yourself. [Interviewee E]

Quote 6-3

I have a problem now with the metals which is a new area. There are different ways of getting the conversion to get an answer, there is some of the advice in the manuals [how to get the conversion] which to use but can go down here [the manual appendix] and see what they recommend. So there are examples [how to get the conversion]. If I don't quite understand the manual, I can see the example. [Interviewee B]

Quote 6-4

If I was coming across run time errors or anything that I didn't know, the first thing I would do I would go to Google and I would try and search up some questions or how do you do this (...) and then try and find the work [around]. If I can't do that, I either email [a participant's colleague] who is like the head of our work or [another colleague] for an answer and they point me in the right direction. [Interviewee D]

Quote 6-5

It starts out very nicely, it tells you how to install it. It has the configuration scripts to configure the way it goes. Tells you what different files you can find in there. How to build it, how to run it. Instructions on simulations if you want to. It's a lot of details here really. Then it's a bit about the model itself, so where it came from. (...) Talks a bit about the compatibility with other similar models, things you can use it for, requirements, more of its history and that's about how it works on the inside. [Interviewee A]

Quote 6-6

looking at things, you can see the flaws, if you have a specific equation that has been simplified and chopped all terms in the highest order of x^2 , (...) and think “I wonder what happen if I put the cubes in or something like that”. (...) On one hand [the software package X] is very easy to work with. (...) But that comes with the price that there is never any reason most of the time to look into the rest of the code and really understand what exactly it is doing. So I probably understand this equation of state routine [from software package Y] a lot better than I understand [the software package X]. [Interviewee A]

Quote 6-7

So this is the main reference for the technique. It’s got a lot of additional information. When you present something, and I’m just as guilty of this as other people, sometimes you take some things as obvious to some other people. I think here there were just a couple of crucial points that weren’t completely obvious. I got something which was pretty much almost right. There is no detail about specific implementations, this is all about rules and probabilities and getting the maths right. You end up with the chunk of maths here. [Interviewee C]

Quote 6-8

I probably should have sent him an email and just say “I’m just getting nonsense out of here”. But then you immediately snowball into collaboration. You’ve got competitors as well, he is a competitor as well, he is a research competitor. (...) I think if I bumped at him at a conference, I would have asked but I didn’t bump into him at that time. I didn’t pick up a phone and have a chat with him, which would be the best solution at the time. If they working on the same problem, you don’t know that, then that may spur them to write the paper quicker. You may end up in a worse position. It shouldn’t be the case. Most of the time it doesn’t happen. Sometimes people are working on things and they discover that other people are working on the same thing and then it’s a bit of a race to finish. It’s not fun. [Interviewee C]

Quote 6-9

I haven’t found them [user mailing lists] that useful to be honest. The one I continually get, I usually delete most of them. I did one time get stuck on how to do something in another program and tried to search for a similar problem, it sort of came up with things about it. But unfortunately what it came up was somebody else having a similar query but no answer. So that wasn’t particularly helpful. On the whole, I don’t find them particularly useful. [Interviewee B]

Chapter 7

Quote 7-1

SciPy is in quick development and (as a consequence) it lacks of documentation. I think the point is that the main developers have too much to do with more important things, while the common user finds it difficult to submit documentation patches. (...) Maybe we could organize a wiki page where to collect new documentation? [mailing list, first half of 2003]

Quote 7-2

There should be an easy way for user-contributed documentation and code-snippets (which will/could be integrated into scipy in an automatic (?) way) (When trying out a command I tend to write a short example to test whether I understood the description correctly. If many users would contribute their mini-examples using a well-defined and simple procedure that could accumulate quite quickly ...) [mailing list, second half of 2005]

Quote 7-3

I can’t remember how many of the functions we have out there. But there is probably a thou-

sand functions, maybe more. We had maybe 30% coverage. And [one of the ASP initiators] wanted to get that covered, I guess, mainly for his PhD students but also just because (...) it's hard for somebody to get into this. He got funding and hired [one of the main developers] and [got] other people to be separating this and they set up a nice website to be able to contribute. [Interviewee A]

Quote 7-4

The basic gist is this: THERE ARE THOUSANDS OF PEOPLE WAITING FOR SCIPY TO REACH CRITICAL MASS! SciPy will reach the open-source jumping-off point when an outsider has the following experience: They google, find us, visit us, learn that they'll be getting, install it trivially, and read a tutorial that in less than 15 minutes has them plotting their own data. In that process, which will take less than 45 minutes total, they must also gain confidence in the solidity and longevity of the software and find a supportive community. [mailing list, first half of 2005]

Quote 7-5

*several people have expressed hesitation in editing the wiki, since they didn't put it up. Folks, it's a *WIKI*! You're *supposed* to edit it! Add, add, add content! All you need is a scipy.org account, which you can make for yourself following the link in the title bar. There is history, so if someone makes a change that isn't popular, we can roll back the change and no harm done. If you want to make major changes to the structure or content posted by others, you can and probably should ask on these lists. But do ask! If it's a good idea, others will agree with you. [mailing list, second half of 2004]*

Quote 7-6

I think it can be done with a special wiki which has a page for every docstring in the sources (not every comment, but every docstring intended for the user). Some script can be run once in a while that will take the docstrings from the wiki and insert them into the sources. Such a system would very significantly lower the barrier for contributing to documentation. Of course, the system is not trivial, it should check the docstrings from the wiki to avoid "injection attacks" etc. etc. (...) Does anybody know if a similar system already exists somewhere? [mailing list, second half of 2006]

Quote 7-7

I was trying to convert some basic analog filters to digital filters, and got hung up by the bilinear() function... which doesn't document its inputs or outputs. Having now figured out how it works, I thought I would suggest an expanded documentation string modeled after that of lfilter() (...). Sorry if this isn't the right place to send documentation suggestions. I will be happy to add more doc strings as I learn how to use this package, if anyone wants them. [mailing list, second half of 2007]

Quote 7-8

*Make a decision on what you want. Stick with it. Don't let it be brought up for discussion. This is pointless bikeshedding that doesn't help improve the documentation effort in the slightest. Everybody else, please just drop it. If you feel like worked up enough to write more emails, *write a docstring instead*. [mailing list, first half of 2008]*

Quote 7-9

SciPy is a community project, and the community should coordinate and review what goes into SciPy. In that vein, this document [the author refers to the email which contains this quote] is a genuine request for your comments, and we will hope to maintain this format in the future so that all voices can be heard. If you are willing to work on any part of this effort, please make your intentions known by adding your name and email (...) to the ASP Wiki (...). [mailing list,

second half of 2004]

Quote 7-10

When there wasn't a person that was actually paid to do what we called flagging the list, flagging the email list and getting out there every day or two "So what are we going to do today. A bunch of us are going to be working on these functions next week and try to finish them all". Just getting out there and keeping the discussion going on the mailing list. When that wasn't happening, things fizzled out in a couple of weeks [Interviewee D]

Quote 7-11

For people who did not have commit rights the barrier to [source code] contribution was much higher. In addition, since we were thinking at the time, well, we really want people who may not even want to deal with the code but they may be able to write the explanation of what the algorithm does, we want them to be able to contribute. [Interviewee G]

Quote 7-12

I am looking to hire someone to write NumPy docs and help coordinate the doc project and its volunteers. The job includes working with me, the doc team, doc volunteers, and developers to: write and review a lot of docs, mainly those that others don't want to write; help define milestones; organize campaigns and volunteer teams to meet them; (...) work with the packaging team to meet their release deadlines; (...) Candidates must be experienced NumPy and SciPy programmers (...). They must also demonstrate their ability to produce excellent docs on the docs.SciPy.org wiki. (...) Ability to take direction, work with and lead a team, and to work for extended periods without direct supervision on a list of assigned tasks are all critical. [mailing list, first half of 2009]

