Abstract

Scientific software, by which we mean application software that has a large computational component, models physical phenomena and provides data for decision support. This can be software that calculates loads on bridges, provides predictions for weather systems, images bone structures for surgical procedures, models subsystems at nuclear generating stations, or processes images from ground-based telescopes.

There is no consensus on what the best practices for the development of scientific software are. We carried out a study at two Canadian universities in which we interviewed scientists and engineers who develop or use scientific software to identify characteristics of current development and usage. Through qualitative analysis, I identified key characteristics of scientific software development and usage and observed correlations between these characteristics.

The results are a collection of observations about how scientific software is documented and designed; the nature of the scientific software lifecycle; the selection of development languages; approaches to testing, especially validation testing; and sources of risk. I also examine concerns scientists have with commercial software they use to determine what quality factors are of interest to them and also which seem to require special trade-offs. I find that scientific software development and use differs fundamentally from development in most other domains.
Acknowledgements

I would like to thank my supervisors, Dr. Diane Kelly and Dr. Terry Shepard, as well as my mother and father, for all their support. Additional thanks to Dr. Kelly for her help with proofreading and her patience.
# Table of Contents

Abstract.................................................................................................................................ii
Acknowledgements..................................................................................................................iii
Table of Contents...................................................................................................................iv
List of Figures..........................................................................................................................vi
List of Tables...........................................................................................................................vii
Chapter 1 Introduction.......................................................................................................... 1
Chapter 2 Background and Related Work ........................................................................... 5
Chapter 3 Research Method................................................................................................. 16
  3.1 Qualitative Data Analysis.............................................................................................. 17
  3.2 Note Taking .................................................................................................................. 23
  3.3 Coding the Data .......................................................................................................... 25
  3.4 Analysis Techniques ................................................................................................. 29
  3.5 Focus Group............................................................................................................... 32
Chapter 4 Observations........................................................................................................ 34
  4.1 Motivations for Development ...................................................................................... 36
  4.2 Requirements Documentation in Scientific Software ............................................... 37
  4.3 The Lifecycle of Scientific Software ........................................................................ 39
  4.4 Design of Scientific Software .................................................................................... 40
  4.5 Development Languages of Scientific Software ....................................................... 42
  4.6 Risks in Scientific Software Development ............................................................... 46
  4.7 Collaboration of Software Engineers and Scientific Developers............................... 48
  4.8 What Scientists Learn About Software Development .............................................. 50
  4.9 Documentation of Code ............................................................................................ 51
  4.10 Data Management for Scientific Software ............................................................... 53
  4.11 Version Control of Scientific Software ...................................................................... 54
  4.12 Code Reviews of Scientific Software ......................................................................... 56
  4.13 Testing Scientific Software – General ....................................................................... 56
  4.14 Testing Scientific Software – Testing Types ............................................................ 58
  4.15 Testing Scientific Software – Validation Testing .................................................... 61
  4.16 Usability Testing and User Documentation of Scientific Software ........................... 66
  4.17 Theory Documentation of Scientific Software ........................................................ 70
  4.18 Quality Factor Tradeoffs in Commercial Scientific Software .................................. 71
List of Figures

Figure 1 Scientific Software Characteristics.................................................................................. 20
List of Tables

Table 1 Interviewee Backgrounds ................................................................................................. 18
Table 2 Static Classification Codes ............................................................................................... 28
Table 3 Dynamic Classification Codes .......................................................................................... 29
Table 4 Example Summary Table for Software Distribution Methods (DISTRIB code)............. 31
Table 5 Oracle Types..................................................................................................................... 62
Chapter 1
Introduction

Scientific software is software with a large computational component that models some physical phenomena and provides data for decision support. Those who develop scientific software may be scientists working in their domain within industry or academia. They are referred to as “professional end user developers”; these are “people who are not professional software developers, but who are members of some identifiable knowledge-rich technical profession developing their own software as a tool for furthering their professional goals” [1]. Alternately, scientific software may be developed by professionals in other domains, such as software engineers, working alongside or in consultation with experts in the scientific domain the software is intended for.

Scientists and engineers (referred to as “scientists” for brevity) developing such software tend to have strong backgrounds in the theoretical models that the software is meant to represent, but they do not usually have a formal background in software engineering techniques. Becoming software engineering experts would not necessarily give them the knowledge they need to develop quality scientific software, since at the moment there is no consensus on the “best practices” for the development of scientific software.

Instead of creating development methodologies specific to different development domains, computer scientists have focused on formulating domain-independent techniques for development since computer science came into its own as an independent discipline in the 1960s, as described by Vessey [2]. Yet, for whatever reason, these methods have not been picked up by the scientific software development community, as is evidenced by Greg Wilson’s observations that scientific
software developers often do not apply even rudimentary software development techniques [3]. Vessey et al [4] argue that those creating software development methodologies cannot persist in ignoring domain details; more domain-dependent methodologies are needed.

One of the key steps to creating software development processes that are specific to a given domain is to determine current practices in the domain. This step is necessary to determine how current practices are working, in what ways they are not working, and to identify gaps in the development process.

There are few studies into current practices in the scientific software domain, making it difficult to move ahead on making recommendations for this development community. To garner qualitative data on scientific software development practices, we interviewed sixteen developers of software in a range of scientific and engineering disciplines.

My thesis is that I can identify practices and surrounding circumstances by which scientists and engineers develop and/or use computational application software as a major component of research in their own field. The goal is to make recommendations regarding these practices to scientists and engineers who develop their own software and to the software engineering community.

The results of this study cover many areas of scientific software development. For example, our study identified three, sometimes overlapping, reasons why scientific software is developed. Though there were many different processes for developing software among our interviewees,
Our study shows that a consistent feedback loop occurs between the development of theory and the development of software.

The results of our study shows a correlation between requirements documentation required for scientific software and the domain separation between developers and users. They also identify three sources of requirements volatility – changes in the theory, changes in the scope of the theory, and quality factors. The results also broadly address design, showing that it is sometimes, but not always, a neglected step in the development process.

Our results also show the relative merits of development languages as described by our interviewees, arguing for the usefulness of procedural languages such as Fortran in domains with heavy computational requirements. We have also heard why object-oriented languages are not always or even usually preferable. We began to focus on the risk management strategies of scientific software developers to explain this preference.

Our study provides commentary showing that validation is the primary type of testing conducted on scientific software. We have identified three main sources of oracles for validation: real-world data, simulated data, and the scientist’s own expertise. Accuracy is an important quality factor for scientific software, and it is found to be a matter of dissatisfaction for several participants in our study. Negative testing was found to be largely neglected (for a definition of negative testing, see the glossary in Appendix C).

Another topic addressed by our study is the degree of satisfaction of our interviewees with open source and commercial software. We found that satisfaction with open source software is mixed
among our interviewees, while commercial software faces difficulties with usability and suitability tradeoffs as well as trust in its black-box calculations.

Section 2 discusses background research pertaining to scientific software development and the importance of developing methodologies in this domain; section 3 provides information about the method we used to collect, organize, and analyze our data; section 4 puts forward observations we have drawn from our study, and section 5 contains conclusions, with references and a glossary following.
Chapter 2

Background and Related Work

Though research into scientific software development is limited, there still exist surveys, discussions and case studies from which we were able to draw some direction for our study. The background literature makes it clear that the type of study we conducted has not been done before on a similarly broad group of interviewees. This also demonstrates that several of the conclusions we have drawn confirm observations from other studies, while others provide new perspectives into the field.

A.F. Granados [5] presents his reasons for conducting a survey of software project management at the 1998 Astronomical Data Analysis Software and Systems (ADASS) conference. Having worked in both research and industrial software groups, first as an astronomer and then at a start-up firm developing image processing software, he observes that scientific software is rarely subjected to a formal development process with strict processes and quality controls. He also observes a divide between best practices commonly accepted by the software engineering community and the methods of professional end-user developers. This adds to other evidence that strongly suggests that scientific software development is presently managed differently from development of other types of software, creating a case for the necessity of our research.

Granados’ [6] description of project management practices indicates the following qualities of the sixty scientific software developers surveyed:

- They work in small teams
- They do not have a degree in computer science
• They follow an iterative, incremental, or “code and fix” development model
• They do a lot of R&D in their scientific programming
• Most of their projects are either delivered late or have no schedule for a delivery date
• Forty-five percent of respondents did not use object-oriented methods to model their software.

This is useful as a comparison point for our study. Granados’ survey differs from our study in terms of characterisation of participants and the type of data collected. Since the survey was given at the ADASS’98 conference, the participants can be assumed to be mostly astronomers. Our study has a broader base across multiple scientific disciplines. Only 8% of participants in Granados’ survey describe themselves as researchers who do some software development; others described their main roles as software developers, software maintainers, or software developers who do some research. Most of the participants in our study would likely describe themselves as researchers doing software development. The mandate of Granados’ survey was confined to project management, whereas our study covers the software development process more broadly. The Granados survey also follows a different methodology than our study. Granados used multiple-choice questionnaires, which means he focused on specific questions he identified as important. We used an unstructured interview to explore issues important to our interviewees.

Wilson et al [7] describe the generalized situation of scientific software developers through several example cases. They describe how non-optimal usage of languages and tools can, at times, seriously affect scientists’ work in ways they may not detect until it is too late. They state that there are two reasons scientist developers tend to approach programming less effectively than they could. One is that they don’t know any better. The second is that they claim not to have the
time to learn yet another programming language, tool, etc. This is exacerbated by what Wilson describes as the inappropriateness of computer science-developed tools for the development of scientific software. This can be used as a point of comparison with our research. As a follow-up to Wilson et al, Jessop et al [8] summarize problems identified by Wilson and others. One is data set manipulation, which can start scientists down a difficult path of needing to learn many tools Jessop et al call “arcane”. A second problem is that scientists often use or add modules onto older software that contains components no one understands. Third, there are significant differences in opinions on the most important subjects for scientists to learn in a hypothetical one-week crash course intended to improve their software development practices. These differences of opinion show that even those scientists who have an appreciation for the magnitude of the problem faced by scientific software developers cannot agree on how to solve it. This is an indication of the seriousness of the problems faced in scientific software development.

Wilson [3] indicates that his opinion on the deficiencies of scientific software development has not changed in over a decade. He claims that the biggest bottleneck in computational science today is still “the one between [scientists’] ears”. He brings to attention the lack of knowledge among other scientists of techniques such as version control and systematic (methodical and repeatable, as opposed to ad-hoc) testing. At the same time, he acknowledges that scientists feel “steamrolled” by computer science fads, from object-oriented programming to agile methods, as they are foisted on the scientific computing community one after the other without any proof of real benefits. He also acknowledges the years it would take for a scientist to sift through the massive information on software development. Add to that observation the time it would take to discover by trial and error what techniques are and are not useful in the scientist’s domain. We compare the results of our research to Wilson’s observations.
Stevenson [9] displays distaste for the advice of software engineers. He is critical of software engineering’s contribution to assuring quality in scientific software, especially its reliance on International Organisation for Standardization (ISO) and Capability Maturity Model (CMM) standards. He finds that the emphasis in software engineering on developing processes that treat code like a manufactured product is incompatible with scientific software, since developers of the latter are more interested in scientific insights than in the code itself. This is further support for the development of domain-dependent methodologies.

In her technical report on supporting scientific software development, Judith Segal [1] makes note of several characteristics of professional end-user developers of scientific software that can be compared with our results:

- Her interviewees regarded themselves as scientists, not software developers, even if they spent much of their time developing software
- Formal software engineering training is limited.
- Documentation of the project is seen by the scientists as insufficient for getting a newcomer up to speed.

Segal promotes agile methodologies as fitting the practices of scientific software developers, indicating the lightweight, iterative nature of development in this domain. Agile methods are frameworks for iterative software development [10]. Her study describes two in-depth cases, one a consulting firm and the other a research group; this thesis studies over a dozen cases, most with smaller development teams than those referenced by Segal, and not all of the cases would be
classified as professional end-user developers like those from Segal’s research. However, Segal’s observations can be compared to our own.

Another case study by Segal [11] addresses a different scenario of scientific software development, wherein the domain expert scientists collaborate with software engineers to produce software. Scientists list requirements that software professionals use as a specification. The specification is checked against the requirements by the scientists, and the software developers develop and test the code. The experiment indicated that providing up-front requirements is anathema to the more iterative way scientists create software. There was difficulty in constructing an understanding between the two groups in order for the requirements document to fulfill its purpose. Amongst our interviewees, there were no collaborations following this paradigm, but the emergent nature of scientific software requirements is a feature in common amongst them.

Segal [12] argues that research into professional end-user development must focus on providing processes or tools that do not demand significant increases in development resources or a rapid shift in current development practices. She argues that software development is seen as an ancillary activity by many professional end-user developers, i.e. creating elegant programs is less important than fulfilling an immediate scientific need. This is in agreement with Stevenson [9] and can be compared to our own observations.

One pilot project conducted at the National Aeronautics and Space Administration (NASA) indicates that adopting practices from an agile method called Extreme Programming (XP), including pair programming and tight iterations, yields productivity boosts [13]. The authors admit that some aspects of the management environment they normally work in do not mesh well
with XP, including the insistence on up-front design, large specifications, the lack of a customer to interact with, and the difficulty in measuring progress quickly on scientific ventures that could take a decade or more to bear fruit. Unfortunately the study was only a pilot program, so there was no information on the interaction between XP and most of these barriers. It is, however, an excellent example demonstrating productivity gains in the pace of development and effective bug trapping when scientists work in tight iterations as opposed to the long development cycles.

The case studies conducted by Carver et al [14] on scientific computing on high end systems make some observations that are similar, though not wholly overlapping, with those in our study. They find that the driver of code development is scientific progress, and other quality factors like performance are only given importance as necessary to meet scientific objectives. This is similar to an observation in our study. Likewise, Carver et al observe that most developers are not computer scientists, that development is iterative, that there is a high staff turnover, that procedural languages dominate as opposed to object-oriented languages, and that verification poses a particular challenge in this domain. The Carver et al study focuses exclusively on software developed for high-end computing systems, and all the projects studied were very large (200-760 thousand lines of code (KLOC)). Our study deals with a breadth of scientific domains and software projects, all of which are smaller than those in Carver et al’s study, but they provide a comparison with our observations.

Other research has studied specific portions of the development process. The formal method model put forward by Kreyman and Parnas [15] emphasizes precise documentation of requirements for scientific software through a tabular representation of mathematical relations. It adapts a four-variable model used in control and information systems for use in scientific
software by adding a fifth variable that separates method-based quantities from environmental quantities. The environmental quantities are representations of real physical phenomena and are thus distinct from method-based quantities. The method-based quantities are dependent on the implementation used and have no corresponding equivalent in physical reality. Kreyman and Parnas show that formal methods could potentially be adopted for use in scientific software, but as in other domains, it is a costly exercise, and it would require a significant shift in resources and processes of the kind that Segal argues against. None of our interviewees used formal methods.

Smith [16][17] proposes a requirements template for scientific software. He argues in its favour that systematic requirements documentation reduces ambiguity, defines the scope of values or situations over which the model is applicable, and forces the domain expert to decide how to handle special cases, such as division by zero. He brings up the difficulty of writing validatable requirements for scientific models because the correct output may not be known \textit{a priori}. Many scientific models rely on continuous mathematics with infinite possible inputs and the potential for singularities, meaning that validating one set of inputs does not mean that a similar set of inputs would produce an acceptable result. He also argues for the use of a symbol table to make variable meanings clear. However, few of our interviewees created a requirements document, and the documentation-heavy approach of this template, which includes a traceability matrix that must be maintained, is not compatible with the way research scientists work (or are willing to work), as described in Segal’s case study.

An International Federation for Information Processing (IFIP) Working Group [18] published research on the architecture of scientific software. This research identifies a simple design architecture comprised of input processing, a computation engine, and output processing that is
common among many scientific software applications. Understanding this baseline structure of inputs feeding into a computation engine that produces outputs is an important point of reference. Unlike this architecture research, our study does not address specific design patterns used by scientific software developers; it is more broadly concerned with where the design step fits in the development process, how much attention is devoted to it, and how satisfactory scientists find the design of software they develop and use.

Commentary on both design and language use in scientific software can be found in Decyk, Norton, and Gardner [19]. They address why Fortran is a dominant language in scientific computing – specifically, that it is “a natural language for expressing science and engineering ideas” with mature, high-performance compilers. They assert that Fortran95’s lack of inheritance or dynamic polymorphism is of little consequence to scientific computing because these features are rarely required in scientific domains, and also note that the aversion of computer scientists toward Fortran is often based on Fortran66 or 77, not the more recent versions of Fortran. This information is critical for retaining an open mind about the usefulness of languages for scientific software development, even those that are not in vogue in other development communities.

A concern raised in Decyk et al’s article is that education in scientific software development lacks tutelage on design patterns relevant to scientific contexts. Norton et al [20] describe their application of the strategy design pattern in Fortran. This work adds to the body of literature suggesting differences in the needs of scientific software development compared with other domains across various phases of the development process. In their commentary, they also note the reluctance of scientific programmers to change their Fortran77 legacy code to Fortran90 or
C++ due to the inevitable risks associated with change. This is an exemplar of the risk aversion of scientific software developers, which this thesis discusses, though in different contexts.

Post and Votta [21] emphasize the importance of verification, validation, and quality management in scientific software projects in their article. They note that computational science is evolving such that it is approaching theory and experimentation in its importance to scientific research, yet it is not treated with the same care and attention by scientists, which is in agreement with Segal’s observations [12]. Post and Votta go further than this to state that their experience with cases in which verification and validation were taken seriously yet still proved to be challenging steps emphasizes the need for new methods, though they go on to state that currently existing methods could be adapted from industry. However, as Wilson [3] attests, whether the scientists on these projects understood existing verification and validation methods to the extent needed to apply them effectively is questionable. Our study is careful to take the education and experience of the scientist developers in software engineering methods into account when drawing our observations and conclusions.

Burnett et al [22] describe research into quality control in end-user programs. The users they were addressing were not scientific software developers, i.e. not professional end-user developers. They proposed several testing techniques to make it easier for end-user programmers to find faults in the absence of a computer science background. Their research argues that since the background, motivation, and interest of end-user developers differ from other development communities, quality control techniques for their applications should also differ; the same argument can be made for many scientific software developers. However, professional end-user developers also differ from end-user programmers in background, motivation, and interest, and
the testing techniques proposed by Burnett et al are intended for spreadsheet applications and are not suitable for scientific code. None of the participants in our study programmed used a spreadsheet.

As described by Kelly et al in [23], members of a software engineering research group worked in collaboration with researchers in Nuclear Engineering at the Royal Military College of Canada to test scientific software developed by nuclear engineers. This research shows that software engineering testing techniques, particularly inspection and regression test suites, can be applied to scientific software successfully through collaboration between the two domains. This research is a useful indicator of how testing of scientific software might be improved. Unlike our research, it deals with one in-depth case. Most of the cases we studied do not include a trained software engineer.

The importance of accuracy to scientific software is underlined by Hatton and Roberts [24] and Hatton’s follow-up [25]. In Hatton and Roberts’ research, over a dozen programs for seismic data processing in the oil industry, all implementing a similar algorithm, were found to deliver drastically different results, with answers becoming more deviant as the amount of computations in a process increased. The accuracy errors in these programs reduced the output from six significant figures to two. For the interpretation of the data to be useful, the data needed an accuracy of three significant figures. The programs were unfit to address the tasks they were intended for. Working on the assumption that Hatton is correct in believing that the seismic industry’s quality controls for software are more mature than most others they have observed, accuracy is likely a critical issue for other scientific software developers as well.
Stevenson [9][26] proposes new ways to measure quality specific to scientific software. He is supportive of accuracy, as well as the use of numerical methods, as ways of measuring the quality of calculations. He proposes that testing methods for scientific software should focus more on numerical analysis, numerical methods, and floating-point computation. He discusses the many sources of implementation-based risk in scientific computing and proposes physical exactness, computability, and bounded errors as three ways to measure quality. Physical exactness as he defines it is the elimination of non-physical assumptions. Computability requires the identification of relationships between variables that cannot be calculated exactly, only approximately. He also requires that bounded error estimates be provided along with any set of results.

The background research presented supports the idea behind this thesis that, as argued in [2] and [4], the most useful advice for scientific software development is likely domain-dependent, and therefore requires further research into current practices in the domain. In general, amongst scientists, programming is often accorded little importance, so little time is spent on software training. This can affect their ability to identify and manage risks. Identifying risks will help in recommendations that could be implemented without rapid change or resource reallocation and that would be more palatable to scientists.
Chapter 3

Research Method

This study was approved by the General Research Ethics Board (GREB) at Queen’s University. The researchers were initially unaware of the requirement for GREB approval; because the study was considered low-risk, consent was granted retroactively for data previously collected. The letter of information form, including a clause allowing interviewees to opt-out of having their data included in our research, was sent to the previous interviewees. The letter of information can be found in Appendix A. Later interviewees were also given copies of the letter of information form and were also allowed to retract their data at any time. In addition, they were verbally read the consent form and consented to be interviewed on tape. The consent form can be found in Appendix B.

The method of gathering and analyzing data is well-documented and repeatable. The method can be broken into several steps, in the following order:

i. The collection of qualitative data through interviews recorded by a digital voice recorder, described in section 3.1;

ii. The summarizing of the digital voice recordings into written data points, described in section 3.2;

iii. Coding of the data and the normalization of the codes, described in section 3.3;

iv. Analysis of the data in each code through observation, discussion and matrix displays, described in section 3.4;

v. The collection of additional commentary through a feedback forum with the interviewees, described in section 3.5.
3.1 Qualitative Data Analysis

Qualitative data was collected through interviews with scientific software developers and users, all of whom are professors or students at either Queen’s University or the Royal Military College of Canada. These interviews are thirty to ninety minutes in length, and each took place at either Queen’s University or Royal Military College. Scientists were usually interviewed individually, but in one case, three interviewees from the same development group were interviewed together, and in another case, two interviewees working on the same software were interviewed together. For most interviews, two researchers were present; for one interview, three researchers were present.

Since we were trying to obtain general information on the state of scientific software development across a variety of scientific and engineering disciplines, we tried to obtain a sample set to reflect this. We interviewed sixteen scientists and engineers; all are developers or users of scientific software. Two are civil engineers, one is a chemist, two are electrical engineers, one is a geographer, one is a computer scientist developing scientific software for medical computing, five are nuclear engineers, three are physicists, and one is a theoretical computer scientist creating scientific software to model finite automata. Eleven of the sixteen are professors, two are PhD students, one is a Masters student, and one is a Bachelors graduate who did scientific software work in industry. Three of the nuclear engineers were interviewed primarily on their usage of scientific software; all others discussed their development of scientific software and usage where applicable. There is a large degree of variation in their software development experience; some are not comfortable coding, while some others have worked on industrial software engineering projects and are very knowledgeable about requirements, design, testing, and version control options available to them. Interviewee background information can be found in Table 1.
Note that interviewees are referred to by number; their names will not be disclosed. The scientific software we gathered data on varies greatly in size from modules of less than 1,000 lines of code to programs of over 100,000 lines of code. Runtime of the software falls into two categories: some run in interactive time and others take hours or days to run. The number of intended users varies; it may be only the developer herself, a handful of people in the same research group, or dozens to thousands of external users.

Another critical difference is the closeness of the knowledge domain of the developer to the knowledge of intended users of the software. In our set of interviewees, this degree of separation can be grouped into four categories:

- **Same domain**: The developers and the intended users have similar specialized scientific backgrounds.
• **Non-specialist scientists:** The intended users may be non-specialists in the same scientific domain as the developers who still possess a fundamental background in the scientific or engineering principles involved in the software. For example, if the developer is a civil engineer specializing in bridge design, the user may be a generalist civil engineer.

• **Scientists in a different domain:** The users may be scientists in a domain different from the developer’s scientific domain. If the developers are computer scientists creating medical software and the users are medical doctors, the users would fall into this category.

• **Non-scientists:** The users are not scientists or engineers, but rather technicians with a limited grasp of the science represented by the software.

Figure 1 illustrates the proportions of software with certain size, purpose, run type, number of users, and user knowledge characteristics.
Note that some interviewees discussed more than one development project, so their software fit into more than one of the discrete categories.

It is important to note that we avoided the pitfall described in [27] of avoiding information that contradicted our preconceived notions or was not in agreement with previously collected data.

We interviewed scientific software developers who had delivered software successfully (in their opinions), developers who were encountering severe problems, and cases in between these extremes. Since we did not usually have a characterization of our interviewees or their software before we interviewed them, we did not have the opportunity to bias our set of interviewees, even on a subconscious level.
All researchers present at a given interview asked questions. There were three researchers: Rebecca Sanders, Diane Kelly, and Terry Shepard. Terry Shepard was present for one interview and did note-taking for a separate interview. The other two researchers were involved in the entire process.

One researcher was always assigned with keeping a digital voice recording of the interviews. Every interview was digitally recorded and transferred to a computer for later analysis, with backup copies kept on CD. The interviewees were always made aware that their comments were being recorded. Since the questions asked during the interviews were never of a sensitive or personal nature and any comments made would not be attributed to them by name, we do not believe that this affected their truthfulness during the interviews. There were brief and infrequent occasions when pertinent comments were made at the end of an interview after the recorder was turned off; these comments were documented in our later note-taking sessions along with the recorded material.

Because the study was not intended to take up a great deal of the interviewees’ time, intrude on their development or research activities, or compromise their intellectual property, we did not regularly examine or collect data from other documented sources, though we did take advantage of examining and discussing design, coding, and testing documentation during one interview when it was presented. In several other interviews, we were presented with demonstrations of the software under discussion.

Since the interviews were intended to be exploratory, we expected that questions important to our research would be emergent in nature [27]. Biasing the interviews with a pre-conceived set of
questions had the potential drawback of preventing the researchers from discovering emergent issues, so the interviews were left unstructured and open-ended, allowing the interviewees’ own experiences to shape the direction of the interviews. At the beginning of each interview, we spent several minutes describing the nature of our research so that the scientists understood what sort of information we were gathering. Then each scientist gave a brief description of his or her work, the purpose of the software as related to his work, and his or her role in his development group. In the interviews dealing primarily with software usage, the scientist described the relationship between his or her work and the scientific software he or she uses.

As the scientist described his or her work and his or her software development and use, we asked questions. Though there were no pre-set questions, we always made sure we covered these topics when we discussed development of scientific software:

- Purpose of the software
- Requirements documentation
- Design
- Development language(s)
- Code documentation
- Version control
- Testing

The first interview did not follow the format typical of subsequent ones. The interview served a dual purpose; not only was it for our research, but it was also part of a presentation to several undergraduate students whom the scientist interviewee was soliciting for advice on how to fix several issues in the software he was developing. He explained the purpose of the software,
described problems his group was encountering in development and deployment, and presented a
demonstration of the installation and operation of the software. Questions were asked during the
presentation by the researchers and the students. The information gathered from this interview
was similar to others; topics that were not covered were addressed in a subsequent interview,
following the usual format, with another member of the same development group.

Some analysis was done during the later stages of information gathering, and this indicated areas
of particular interest to us. As a result, later interviews became naturally more focused; while we
still endeavored to cover each subject area broadly, we also built on data from previous
interviews by asking more specifically about issues as they arose from our analysis. This allowed
us to gather data consistently in key areas that were yielding results.

3.2 Note Taking

After each interview, the digital voice recording was summarized into written notes were be
referenced throughout the subsequent analysis steps. This is done as follows.

Since note-taking is inevitably subjective and selective [27], we, the researchers, felt it was
important that we agree on the level of detail and type of information to write down, as well as
the format in which the notes were to be taken. For the first interview, three separate sets of notes
were taken and subjectively compared for similarity and depth of content. We agreed upon a
format for the notes similar to the format one of the researchers used, and the three sets of notes
were combined into one set in that format. From this point forward only two researchers – Diane
Kelly and Rebecca Sanders – were involved.
The first interview was transcribed in its entirety, from which point-form notes were taken by one of the researchers. Another of the researchers took notes as she listened to the digital voice recording without a transcription step. Upon comparison, transcription did not seem to add significantly to the quality of the notes and was extremely time-consuming, so this step was dropped for subsequent interviews. For all other interviews, researchers listened to the digital voice recording of the interview and made point-form notes without a transcription step. On rare occasions, if any passage was hard to understand due either to poor recording quality or the complexity of the topics under discussion, it was transcribed to make parsing the passage into point-form notes easier.

To further ensure that the depth of notes and the type of information written down would be consistent and agreeable to all the researchers, several other sets of notes were subjected to the process of double note-taking. This involved us taking notes independently from the same digital voice recording. When both sets of notes were complete, one of us examined them for similarities and differences. This was done through color coding: if two notes from both note-taking sessions are similar, they were color-coded green and placed beside each other in a table; if one note did not have a comparable note in the other set of notes, it was colored blue and placed alone on one side of the table; if a note contradicted another note, they were placed beside each other in a table and colored red. Since the granularity of the notes sometimes differed, each note might have more than one note matching or contradicting it. After this process, the two of us who did the note-taking went through the notes and discussed the differences, reaching an agreement over any contradictions in their notes and deciding on an appropriate level of granularity to aim for in future sessions. This served as a reliability check on note-taking.
When we were satisfied with our interpretations of the digital voice recordings were similar enough, the remaining sets of notes were taken by only one of us.

### 3.3 Coding the Data

As interviews continued to be conducted and notes were taken from the digital voice recordings, we began the process of focusing and bounding our collection of qualitative data. This was done through a well-defined technique called coding [27]. Codes are a summarizing notation; they make analysis more efficient and effective by both grouping notes on similar subjects so they can be compared during the later phase of pattern-finding and demonstrating where notes are incomplete.

Our codes are descriptive in nature. They do not entail any interpretation or analysis of the data beforehand; they categorize which classes of data the note is relevant to. Each code is two to eight letters in length and is semantically close to the term or concept it represents to enable quick interpretation, as recommended in [27]. For example, notes related to testing are coded as TEST.

Before coding could begin, we needed to create a list of codes that were well-understood by both of us. Since our research was exploratory, we did not have an initial set of codes determined beforehand. Several iterations of the codes were expected as field research continued; codes are often revised in case studies because the research does not fit them or because the structure they create does not fit with empirical observations [27]. Midway through our research, we brainstormed two sets of codes individually, which were compared and contrasted until an initial set of codes was agreed upon. Coding was done using a table with the notes placed in the right
column and a space for inserting relevant codes on the left. Each note was labeled with one or more codes.

Using the initial set of codes, we engaged in a process called check-coding, by which we independently code the same set of interview notes and compare their results [27]. This was done to sharpen the definitions and boundaries of each code and to serve as a check on the consistency of the coding. The comparison was done through color coding similar to that of the note taking checks; green indicated that the code was the same in both interviews and red indicated that they were different.

A disagreement between us sometimes meant that the definition of a given code had to be expanded, contracted, or a new code had to be created. At other times, one of us would agree that the other’s coding was more appropriate, which served as a consistency correction. A subset of the interviews were check-coded; the rest were used as a control set.

During check-coding, we decided on a three-category system for codes: static scientist, static software, and dynamic software codes. Static scientist codes pertained to background information on the scientist; static software codes pertained to characteristics of the scientific software discussed; dynamic software codes pertained to the development or usage of scientific software.

When both of us were satisfied that the coding was acceptably consistent and the codes had evolved enough to encompass all the interview data in a useful way, I coded the rest of the interviews. Satisfaction with the check-coding process was partially measured by the percentage of coding that was the same between us, but since some interviews and some data points were
inherently harder to code than others, this was not a consistently useful metric without also using our judgment. Finding an appropriate coding schema during this phase was an iterative process by which we identified new codes, broke up existing codes into smaller fragments, added codes for data not covered satisfactorily by other codes, or combined codes into more complete categories.

Each time a new coding schema was made to replace the old one, all notes coded before had to be re-coded with the new schema. Most of the re-coding was done by me after the check-coding phase. The codes changed little after check-coding concluded, though some minor changes did occur, such as changing the names of categories to be more readily understandable or more easily accessible through searching. Since the codes were sorted into three categories – static scientist codes, static software codes, and dynamic software codes – it was sometimes also necessary to move a code to another category or add a code with a similar meaning to a different category depending on the context of the notes. For example, one code change that was necessary was for the classification of statements related to hardware. Data on hardware was initially considered to be dynamic, since it pertained to opinions on development and use, but data on the hardware platforms used by the software being discussed was static. As a result, two different hardware classifications were necessary. For a full listing of final codes with definitions, see Figure 2 for static codes and Figure 3 for dynamic codes.
<table>
<thead>
<tr>
<th>Scientists</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACK</td>
</tr>
<tr>
<td>DATE</td>
</tr>
<tr>
<td>DECVYRS</td>
</tr>
<tr>
<td>FIELD</td>
</tr>
<tr>
<td>JOB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE</td>
</tr>
<tr>
<td>DATASIZE</td>
</tr>
<tr>
<td>DATASRC</td>
</tr>
<tr>
<td>DEVSIZE</td>
</tr>
<tr>
<td>HARDENV</td>
</tr>
<tr>
<td>NAMESOFT</td>
</tr>
<tr>
<td>NUMUSER</td>
</tr>
<tr>
<td>PURPOSE</td>
</tr>
<tr>
<td>RUNTIME</td>
</tr>
<tr>
<td>SEDEV</td>
</tr>
<tr>
<td>SOFTENV</td>
</tr>
<tr>
<td>SOFTLANG</td>
</tr>
<tr>
<td>SOFTSIZE</td>
</tr>
<tr>
<td>STABLE</td>
</tr>
<tr>
<td>STATUS</td>
</tr>
<tr>
<td>USERCHAR</td>
</tr>
<tr>
<td>YEAR</td>
</tr>
</tbody>
</table>

Table 2 Static Classification Codes
Analysis of the data happened in both casual and systematic ways. We discussed our impressions between interviews, using our observations to draw hypotheses about the data that could then be tested through systematic analysis. An example of this is language use among the interviewees, which we discussed often in general terms before and during the early phases of analysis in order to determine a hypothesis to investigate. These discussions occurred in an ad-hoc way and were usually undocumented. They were used only as a precursor to systematic analysis. Conclusions were not drawn directly from casual observations. Observations had to be justified by the use of a cross-case display or a summary table to show all the relevant information.
We used cross-case displays (see [27]) as our primary systematic analysis method in order to compare and contrast data gathered from all, or a subset of, the interviewees. Given the exploratory, breadth-focused nature of our case study, it was important that our displays not impose an order that did not exist and that they allow us to contrast several codes of data for each interviewee at once in raw text form.

For these reasons, we used a loosely organized and flexible display format called partially ordered meta-matrices [27]. Typically, each column contains a subset of data from a particular code, while each row corresponds to an interviewee number. We also used another type of partially ordered display called a summary table; as the name suggests, this summarizes information in a category. For example, Table 2 is a summary table grouping the different types of software distribution methods used by different interviewees.
<table>
<thead>
<tr>
<th>Closed source, not distributed for free</th>
<th>Closed source, distribute on request</th>
<th>Source code distributed, but not for free</th>
<th>Open source, distribute on request</th>
</tr>
</thead>
<tbody>
<tr>
<td>Want to patent the software and distribute it to reactors around the world [S01-02]</td>
<td>Software is downloadable from her website [S07]</td>
<td>Don’t really share their models, especially since he’s doing research on new models, so he’s not going to use other peoples’ models [S13]</td>
<td>Software isn’t disseminated outside university; others implement their own version to verify the results [S03]</td>
</tr>
<tr>
<td>Thinks licensing is becoming an “issue” with scientific software [S09-11]</td>
<td>U of T software won’t share its source code, but her old professor will give her some of it sparingly when she has a particular problem, but she has to swear she won’t “divulge” it [S07]</td>
<td>People have asked for his models, he’ll give them to him if they bring something to the table, quid-pro-quo [S13]</td>
<td>Would give code to other scientists if they asked [S03]</td>
</tr>
<tr>
<td>Software is distributed to hospitals [S12]</td>
<td>Thinks the reason people don’t want to share code is that it’s happening in academic context, how she gets rated has to do with what she produces, so if the software gets used a lot, she wants credit [S07]</td>
<td>Says that academic way of treating source code is competitive, not collaborative [S07]</td>
<td>Sometimes shares code with other research groups when asked for it [S04]</td>
</tr>
<tr>
<td>Never owned the software he produced, the company he worked for owned it; he didn’t want to be liable for maintaining/supporting the software [S13]</td>
<td>Says that academic way of treating source code is competitive, not collaborative [S07]</td>
<td>Program he used was closed-source but free – dev sends it to you if you email him a request [S08]</td>
<td>Send the code to anyone who asks for it [S16]</td>
</tr>
<tr>
<td>Software is downloadable from her website [S07]</td>
<td>Software given to military [S13-14]</td>
<td>Software given to military [S13-14]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sent software to a library in Moncton – impossible to track who gets it after that [S13-14]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4 Example Summary Table for Software Distribution Methods (DISTRIB code)

During the process of creating displays, we found it efficient to sort the data by its code to make all the data in each code more accessible. This did not involve any recoding of the data. For
comments that fit into more than one code, copies were made for each code. This split the data into three files for each interviewee: one for static scientist codes, one for static software codes, and one for dynamic software codes.

We found it useful to begin with several meta-matrices including codes that either had been casually observed to relate to each other during our discussions or had some theme in common, such as all codes relating to dataset management, usage, size, etc. In the case of quality factors, we created a meta-matrix of all the different quality factors discussed in the interviews. Often particular themes of interest emerged, or some of the codes could be pruned due to either an apparent lack of correlation or a lack of sufficient data in the code. In these cases, further matrices and summary tables were made by partitioning the data into smaller sub-groups or clustering related codes to make contrasts between them clearer.

We used these displays to find patterns and themes among the data. Clustering based on development and usage factors common among several cases was also useful in determining patterns. For example, clustering the interviewees who were dealing with external user groups made it easier to note which quality factor concerns affected them compared to those who had no external users to satisfy. The observations in Section 4 discuss the patterns and themes that were found.

3.5 Focus Group

After coming to several preliminary conclusions through systematic analysis, we decided to take our results back to the community of developers we had interviewed to gather their impressions and any final commentary they wanted to impart. This also served as a check on our conclusions.
to ensure that they made sense to the practitioners themselves. Four of the interviewees attended; these interviewees are identified as S03, S04, S09, and S14.

After the presentation of our research, approximately twenty minutes were taken up by discussion of the interviewees’ impressions as they asked us questions and participated in a broad discussion. The discussion and information gathered in the focus group did not contradict any conclusions we had already drawn, and in fact it complimented data from other interviewees. The focus group participants agreed that our observations were broadly correct, and some of the observations spurred discussion about important design, accuracy, and transparency issues that had not percolated during the interviews. One of us took handwritten notes during and immediately after the focus group. Our analyses, particularly our relevant data displays, were adjusted to include this data.
Chapter 4

Observations

There are twenty-one observations, divided by subject area. This section includes:

i. Why scientists develop software, described in section 4.1;

ii. The treatment of requirements in scientific software, including the general lack of requirements documentation unless such documentation is mandated and the importance of the theory as a primary source of requirements, described in section 4.2;

iii. The typical development cycle for scientific software, including the presence of Royce’s “do it twice” [28] development process, described in section 4.3;

iv. How scientific software is often not consciously designed, including the creation of “behemoth” programs, and the distinction between the computational engine and the user interface in design, along with factors influencing the emphasis put upon each, described in section 4.4;

v. The preference for procedural languages over object-oriented languages, with the exception of some engineering fields, and how language choice affects risk, described in section 4.5;

vi. The sources of risk in scientific software development, described in section 4.6;

vii. The perceived challenges and successes of software developers working alongside domain experts, described in section 4.7;

viii. How scientists gain software engineering knowledge, described in section 4.8;
ix. The paucity of documentation in and of the code itself, the danger of this given the loss of institutional memory, and the components of code documentation that are deemed most useful by scientists, described in section 4.9;

x. Data management challenges and approaches, described in section 4.10;

xi. The varied approaches to version control among our interviewees, described in section 4.11;

xii. The lack of code reviews, described in section 4.12;

xiii. General impressions of testing, described in section 4.13;

xiv. Types of testing that are done on scientific software and types of testing that are notably absent or weak, described in section 4.14;

xv. Validation testing, including the importance of oracles to scientific software testing and an assessment of their adequacy, described in section 4.15;

xvi. Approaches used by scientific software developers for addressing usability risks through usability testing and user documentation, and the correlation between the risks associated with usability and the similarity of the users’ scientific backgrounds to those of the developers, described in section 4.16;

xvii. The importance of theory documentation as software documentation, described in section 4.17;

xviii. The quality factor tradeoff between suitability and usability in commercial software, described in section 4.18;

xix. Concerns about accuracy and transparency in commercial software among scientific users, described in section 4.19;

xx. The low usage of open-source software among our interviewees, described in section 4.20;
4.1 Motivations for Development

The question of why scientific software is developed in the first place is fundamental. Understanding development goals is crucial for determining which improvements will contribute most to meeting those goals. I found three reasons why our interviewees developed software:

- **Research:** The scientists we interviewed were all researchers in their own application fields. Software is a tool to demonstrate that the theory works. It is used to test their models and to provide evidence for research publications.
- **Training:** Some scientific software tools are developed to be used in the classroom and the lab to train our next generation of researchers, scientists and engineers.
- **External decision support:** Scientific software is sometimes moved from the research lab to external use, or is developed solely for external use from preexisting theory. It provides data to support other professionals in decision-making. Our interviewees had software that had moved to the nuclear industry, to the military, and to the health sector.

These purposes are not mutually exclusive. S01 and S02, who work in the same development group, are in the process of commercializing their software for the nuclear industry. This software is based on their own evolving research models. S07 is creating software for solving systems of finite automata; she intends to use the software in her research and as a teaching aid for courses in her field. Nine of the thirteen developers we interviewed (the other three interviewees were primarily interviewed on their use of scientific software and their validation testing) were
developing software for the purpose of research. One was developing software for training highly
qualified personnel (several other interviewees used commercial scientific software for training
purposes), and seven developed software for external decision support.

4.2 Requirements Documentation in Scientific Software

The degree to which requirements are documented varied among our interviewees and is
influenced by the characteristics of the software’s users and the regulatory climate. For research
efforts in which regulation is not a concern and the developers themselves (or specialists in their
field) are the intended users, theory documentation is the most important, and often the only,
source of requirements. In this case, it is common for unwritten requirements to be an
understanding that the software implements the theoretical model of the scientist. The theory
could change during the course of development, changing the unwritten requirements.

Documenting requirements in a way commonly understood in software engineering takes on
more importance where there are regulatory bodies that necessitate the production of documents.
In such cases, regulations may require that the developers meet some guidelines for formal
specification of requirements before the software is accepted or deployed. None of our
interviewees created a traditional, formal requirements specification unless they were forced to do
so to comply with military or industrial regulations – and these specifications were always written
after the software was almost complete. In the case of S14 and S15, they talked of the difficulty in
complying with regulations that did not fit the scientific software paradigm. For example, the
regulations required a number for “availability of the software”. This number was difficult to
estimate and its importance to this example of scientific software was doubtful. Despite such
instances, S14 and S15 stated that the process of creating a requirements document, even after the fact, was useful for organizing their thoughts.

When the backgrounds of the software’s developers and its intended users differ – such as when the users are scientists in different fields or when the users are non-scientists – a trust-based relationship exists between the scientists and their customers. In all cases with an external customer, the customers ceded decisions about the software to the scientists, acknowledging their expertise in the scientific field. In a more nuanced case of trust, the head (S07) of a research group who had less knowledge of software development than her students played the role of the customer: she stipulated some high-level requirements, such as the requirement that her finite automata software contain an undo/redo feature, but allowed the developers, both students and several hired software developers, to decide other requirements to include.

Trust appeared in the case of S12 along with some active requirements elicitation. In this case, two fields of expertise were being combined in the software: that of medical science and that of imaging theories from computing. The gulf between the implementation of the imaging methods and the doctor’s requirements was bridged through the use of interactive storyboards implemented in Microsoft PowerPoint by the software developers. The storyboards were presented to the doctors, discussed with them, and iterated using their feedback until both the software developers and doctors were satisfied with them. Since the medical field is one with regulations that require a requirements specification document, a formal requirements document was created as well as the storyboards.
4.3 The Lifecycle of Scientific Software

One common point shared by all of our interviewees was an iterative development process. This process did not change regardless of whether the theory – which, as described in the previous section, is often the main source of requirements – was already well-defined or still being researched and refined.

Sometimes the theory being implemented is still a work in progress. Whether the theory is correct or incorrect is unknown, or at least unverified. In this case, the software is often used to both demonstrate and test the theory. This results in a highly iterative feedback process between the theory and the code.

Even when the theory is mature and well-understood, I found that an iterative approach is used. In such cases, rather than the theory changing, the scope of the theory reflected in the code changes as the software development proceeds. For example, S14 and S15’s civil engineering software grew in scope as S14 added the capability to classify different types of bridges.

Royce’s “do it twice” [28] development process showed up in two of our interviews. The expansion of scope of the civil engineering application of S14 and S15 caused S14, the developer on the team, to reassess the structure of the software. In the second year of development, he decided to completely redesign the software. He spoke with enthusiasm about how the redesign simplified the addition of new features. S12, the scientist developing software for the medical field, made a clear distinction between the research version of the software developed in his research group and the version put into the hands of the medical personnel. He had developed a two-step process. Researchers explored their theory and wrote their software
without any formalized software development process. Once the research software was assessed as ready, it was rewritten by a professional developer, with the degree of rewriting necessary depending on the quality of the original research version of the software.

Another case involved an extended development process with multiple prototypes. For the past ten years, S07 had Masters students develop her finite automata software, but each time they did so, she claimed it became “obsolete” too quickly. By “obsolete”, she claimed that it “didn’t become usable enough soon enough”, but how she made this assessment is unclear. From her comments, it seems as though the decision was made based on her perception of whether the development language was obsolete. In the case of the original version, it was written in Pascal, which she saw as passé by comparison with the current language of the project, which is Java. After several versions, she decided to take the then current version and have it more seriously developed, work that her research group is still doing after several years.

A long development process, spanning several years or even decades, was not uncommon among our interviewees. In contrast, some code was essentially thrown away when the objective was reached – for example, publishing a paper based on the theory reflected or proven by the software. Thus the lifespan of scientific software tends to be either very long or very short.

4.4 Design of Scientific Software

Some of our interviewees did not have a conscious design phase in their development process. When design was attempted, approaches varied widely.
In some cases, the effect of a lack of design, or poor design, was clear. S03 stated during the focus group that he tended to “skip” the design phase. He later hired a software developer to rewrite parts of his program to improve time performance. In the case of S02, the lack of a design step in his process made it difficult for him to transfer a previous software project from research to commercialization. He initially created an application to demonstrate his models for the purpose of publishing a few papers. When industry became interested in his code, he expressed regret that it was not written in a more “structured” manner.

In some cases, scientists mentioned adding modules to what was referred to in several interviews as either “behemoth” or “monster” programs. These were scientific software applications that had modules tacked onto them over a long period of time by many different researchers. S04’s approach was to act as the architect for modules being added onto her software; she told her students what she expected the interfaces of their modules to look like so they could be attached to the rest of the program.

S12 used a mature design process. The applications developed by his lab often required common features such as logging, undo, acquiring images, and saving changes. His group used an architectural framework for its applications that included support for these and other features. Another mature design process was undertaken by S14 and S15; their bridge program was one of those that followed the “do it twice” model of redesign. The redesigned software made it easy to add new bridges without overhauling existing code. They also used Rational Rose, an object-oriented software design tool, to model their architecture, with some success in their view.
Some of our interviewees tended to speak about their software as if it had two distinct parts. In reality, the software may not have been written with this distinction in mind. One part is the computational engine; this encompasses the implementation of the scientist’s theoretical models. The second part is the user interface.

The computational engine was always considered to be of great importance to the developers – if the computational engine did not work, the software was useless to the scientist. The attention paid to the user interface varied greatly depending on how important usability was to the scientist’s perception of the success of the project and how different the intended users were from the developers.

**4.5 Development Languages of Scientific Software**

At the implementation level, the developers interviewed were all willing to discuss software languages. When we discussed language preferences, we encountered decided opinions on the pros and cons of different languages. Some were supportive of procedural languages and unhappy with what they perceived as a push by external forces towards object-oriented languages, while others used object-oriented languages for various reasons and with varying degrees of success.

The primary procedural languages discussed by our interviewees were Fortran and C. The object-oriented languages discussed were C++, Java, and Visual Basic. Other languages like Maple, Matlab, Python, Perl and shell scripting were also mentioned. S09, S10, and S11, all nuclear engineering students interviewed together, agreed that Maple was very similar to C since that is the language it is based on, but Maple includes higher-level functions that they find useful.
While it was acknowledged that Matlab was too slow to be a computation workhorse (S03 claimed an eighty time speed increase when he converted his code from Matlab to C), it was used by S02 for pre- and post-processing of data and by S12 for some prototyping by his students. S03 was very positive about his experience with Matlab, stating that it encouraged good documentation practices and early testing. S06, who uses a large variety of languages, mentioned that she frequently worked with Perl and shell script. S12 had some of his students use Python to create prototypes for determining the feasibility of their research.

Fortran was the most common languages used by our interviewees, and their comments indicated that it was pervasive among several scientific fields. S02 stated that most code in the nuclear industry is developed in Fortran. S05 called it a “dirty secret” that most scientific programming is still done in Fortran. His choice of words in calling it “dirty” seemed due to his perception that Fortran is not looked upon well by developers in other communities. Several other interviewees expressed similar sentiments that Fortran is the predominant language in their field. Aside from its pervasiveness, interviewees cited its math libraries and its “convenient” features (what those features are was not explored with the interviewees) as reasons for selecting Fortran. One negative comment received about Fortran was from S02, who said that he found all the changes in Fortran hard to keep track of.

C was another procedural language used by some of our interviewees, with their reasoning being the same as that for picking a procedural language in general – it meshed with the way they think. S03 stated that his favorite language to program in was C, and described himself as “moderately happy” with it. S06 and S16 also used C; in the case of S06, C was one of several languages she used. S05 described C as “Fortran without some of the convenient features”, and this seemed to
agree with other comments related to the lack of parallelized math libraries (which Fortran has) and the perception of Fortran as being the “number-crunching” language (as S06 put it) in general. In order to take advantage of some of the features of Fortran that C lacked, some of S03’s code was in the process of being converted to Fortran.

Some of our interviewees chose to develop their software in C++. The reasons given differed greatly. S01 and S02 chose to develop their software in Visual C++ despite the fact that most code in the nuclear industry is written in Fortran. S02’s reasoning was that he felt that it was important to keep up with the latest developments in languages, and his team had the most experience in Visual C++. He also stated that it was easier to put graphical user interfaces onto his programs in Visual C++ than in procedural languages. The differences in Visual C++’s optimization on different platforms caused them significant trouble: their numerical results were different on different platforms. S12 chose to develop software in C++ because his team used toolkits written in C++ and because it had enough software controls for his purpose. S13 used C++ to implement his models using a commercial software package that was written in C++. On the other side of the argument, S03, who wrote his software primarily in C, expressed a negative impression of C++. The “extras” in C++, by which he seemed to mean some of the object-oriented features of the language, forced him to write more code than he would need to in C to accomplish the same objective. He found this frustrating.

Two interviewees used Java. These were the theoretical computer science development group working on software to model finite automata (S07) and the electrical engineer working on inductor design in industry (S08). The latter cited two reasons for developing in Java: JavaCC, an
open-source parser generator for Java, caused him to choose Java over C++, and Java was the development language he had been taught in school.

Two development projects were done in Visual Basic. These projects were discussed by S13, S14, and S15. S14 and S15 were working on the same software, with an electrical engineer doing the development and a civil engineer taking charge of the theory. This project was in the civil engineering domain, as was the project worked on by S13. In both cases, the code developers seemed pleased with their choice of language. Comments they made on Visual Basic included:

- Visual Basic’s interface took care of bad habits/ignorance of good practices [S13]
- The civil engineer could read Visual Basic code (a little) to see his calculations in the code [S14-S15]
- The interface facility of Visual Basic was helpful [S14-15]
- The difference between object-oriented approaches versus procedural approaches in Fortran required him to adjust his way of thinking [S13]

The subject of procedural languages versus object-oriented languages generated heated responses. On the side of procedural languages, some interviewees argued that a great deal of legacy code they need to work with is written in procedural languages, especially Fortran. They found procedural languages better for number-crunching and input/output. The most common comment on languages was that procedural languages fit the way our interviewees think and the way they work. S03 stated that “OO doesn’t buy me anything”, and S05, after deciding to learn Java, couldn’t see how it could fit into his research.
The feedback on object-oriented languages was not all negative. S13 stated that he thinks object-oriented programming is well suited to situations in which a program “goes from one window to another”. S14 and S15 found that a physical component of a bridge corresponds well to an object in their program, so object-oriented programming was a good fit for their domain. Others, such as S08 and S12, appreciated some of the tools available for use with the object-oriented languages they chose.

4.6 Risks in Scientific Software Development

We did not ask our interviewees about risk management specifically, nor does our coding scheme include a code for risk management. Though we did not set out to discover information on risk management, the observations in 4.1-4.5 all indicated specific areas of risk common to many of the scientific applications we discussed:

(i) the underlying theory for the software,

(ii) the software implementation of that theory, and

(iii) users’ operation of the software.

In section 4.1, I presented the motivations for creating scientific software. When the motivation is largely research-based, risks (i) and (ii) are the critical concerns. For teaching or commercial purposes, risks (i) and (ii) are also present but are joined by risk (iii), which is prevalent when scientific software is intended to be used outside the research group developing the software.

Requirements documentation is also highly coupled to risk:
• Risks from the underlying theory of the software are addressed through the documentation of the theory itself, with the theory being the primary source of requirements.

• Detailed requirements documents were only created when regulation or usage risk made it important, and a detailed requirements elicitation step was linked closely to usage risk.

Those projects that encountered regulatory need for a requirements document were also those with external users. The requirements documentation they provided was meant to address risks from the software implementation of the theory and the user’s operation of the software, but as discussed in section 4.2, whether this exercise was of value to the developers or customers is questionable.

The iterative development cycle described in section 4.3 contributes to addressing the combination of these risks. When the theory to be implemented changes due to the natural evolution of research, or when the scope of the theory to be implemented changes, this volatility results in further iterations in the software cycle. As testing turns up bugs in the software, these risks to the implementation of the theory are also addressed by further iterations. User interface testing to address the third risk caused iteration of some interviewees’ development efforts as they added help files or redesigned interfaces to address the concerns of users.

As discussed in 4.4, scientific software applications include a computational engine and a user interface component that are often conceptually separated. The computational engine embodies the scientist’s theory and models and comprises the two risks related to underlying theory and
implementation. The user interface comprises the third risk related to the operation of the software. This risk is increased if the scientist has limited knowledge of the user’s needs.

In 4.5, I noted that scientists choose their programming language based partially on the conceptual organization of their domain. In some domains, scientists think about their models in procedural terms. In civil engineering, our interviewees found that the object-oriented approach effectively represent the physical objects in their domain. Choosing a language that adds the least conceptual complexity to development – the language in which it is easiest to see the theory in the code – adds less risk. This was only one portion of language selection and this consideration was not accounted for in all cases, but it was the most prevalent answer we received about how scientists chose between procedural and object-oriented languages. The preference for the time-tested math libraries in Fortran can also be seen as a risk-minimization choice.

4.7 Collaboration of Software Engineers and Scientific Developers

Attitudes toward the idea of collaborating with a software engineer were mixed. Some believed that the collaboration could add value in specific areas. Interviewees were also wary of someone who is not a scientific expert interfering with their models.

The specific areas in which software engineers were believed to be capable of adding value varied among the interviewees. S02 believed that having a software engineering expert in his development group would be helpful. However, it should be noted that the specific word he used was “programmer”, not software engineer; S02’s impression of software engineering was very programming-focused. S03, who was working with a software engineer to convert some of his code from C++ to Fortran, believed that a software engineer could help his group with data
management and user interface design. S07 had hired several software engineers to work on her program in the past and viewed it as a positive experience.

There was also a marked aversion to allowing software engineers to interfere in the computational engine of the program. S03, though he allowed that software engineers could be useful in some aspects, was very clear in stating that the arrangement would not work out if the software engineer didn’t agree to keep his hands off his algorithms, as embodied by the computational engine. S07 seemed to concur with this view; she had her grad students working on the computation engine of her finite automata software and had previously hired software engineers to develop the user interface. She noted that differences in practices between a software engineer and one member of the student development group (all of whom were computing students) had created tension.

S13 tried to work with a software engineer in the past, but found that it was faster to develop the program himself because the software engineer didn’t understand the scientific problem the software was meant to address. Similarly, S16 said that a software engineer without domain knowledge would not be useful because there would be no common understanding and they wouldn’t speak the same language. S06 couldn’t think of any way a software engineer could be of help to her development.

Collaboration between software experts and scientific experts works better when the software expert is willing to gain knowledge of the scientific domain. Our set of interviewees contained one example of collaboration between an electrical engineer in the role of the software development expert and a civil engineer in the role of the domain expert (S14 and S15). The civil
engineer taught the electrical engineer how to do the civil engineering calculations by hand, and they established precise terminology and vocabulary to build common ground between them. The civil engineer stated that by the end of the development project, the electrical engineer knew more about classifying bridges (the purpose of the software) than most civil engineers. The electrical engineer did all of the design and coding, and they reviewed sections of the code together.

Another example of a software expert who gained knowledge in a scientific domain was S12, a computing expert who developed software for decision support to be used during orthopedic surgery. He spent hundreds of hours in the operating room observing orthopedic surgeries to gain a better understanding of the domain, which allowed him to identify many unique requirements in the field that his software needs to take into account.

4.8 What Scientists Learn About Software Development

An attitude prevalent in the scientific software development community is that they want to get their science done without worrying about software engineering issues. S05 said that his goal is to implement the science, not to write elegant or optimal code. S01 said that he didn’t have time to learn computer science, yet his lack of knowledge of software engineering was a source of many of his problems, such as the difficulty the group was having creating an installer for their software and their unrealistic estimation of the time and effort to complete development activities. S08 got the distinct impression from his time in industry with engineers that they were “disdainful” of software engineering methods.

Not all interviewees shared this view. S06 was active in seeking out software development knowledge that was useful to her, and several others had industrial software development
experience that had increased their knowledge of software engineering concerns and methods. S16 said that his scientific domain recognized the necessity of applying software engineering methods and that a domain expert with more software engineering knowledge than usual was very valuable. However, even in these cases, as discussed in 4.1, the purpose of their software development was always tightly linked with the computational engine; the implementation of the science is almost always the overriding priority.

4.9 Documentation of Code

The code documentation of scientific software varied among the interviewees. Some documented their code well; others admitted that they did not and that the documentation standards used made it nearly impossible to comprehend someone else’s code.

Those who documented their code well mentioned specific documentation practices that they found valuable. One is sensible variable naming that allows them to correlate a variable in the code to the scientific variable it represents; another is commenting the code so that it can be understood by another developer without needing to speak with the code developer. S01 stated that his code included many comments; the comments included the scientific variable names for variables in the code and references to supporting documentation. S08 stated that he kept his variable names close to engineering conventions.

S13 said that he made his variable names long and human-readable; he did this because it was important for him to understand his code months or years after it was written. He believes he was successful at making his code human-readable based on the fact that others have added to his code without needing to speak to him. S14 and S15 also favored long, descriptive variable names.
to make their code easier to read. S06 finds it important to make sure variable names are short and descriptive, and she expects her students to comment their code to a standard such that she can understand the code after they’ve graduated; S07 enforces a similar standard.

Three of the interviewees mentioned specific tools they used for code documentation. S08 documented his code with Javadoc, and S12’s development group used Doxygen. S14 used a tool to keep track of number of lines of code in his software and the percentage of comment lines.

Most of the other interviewees felt that the code documentation practices in their development groups, or other development groups they had observed, were ad-hoc. The documentation in S03’s group is sporadic; though he encourages comments in code, he does not enforce it. S04 said that the documentation practices of her students were so poor that she had devised a way to work around them by specifying an interface for their modules that they have to provide and treating their code as black box.

Some interviewees agreed that documentation of code was so bad that they wouldn’t expect themselves or their students to be able to understand someone else’s code. S04 encourages students to write code from scratch instead of using other people’s code because she thinks it’s probably easier for them than trying to determine how someone else’s code works. S16 stated that, due to a combination of laziness and protectiveness of their code, code authors in his development group created cryptic, inadequate code documentation that could only be figured out by someone very smart. He compared such code documentation to a Russian textbook – concise and hard to understand unless one already knows the material.
S05 concurred that he would never try to significantly modify existing code unless he could collaborate with the code author because the documentation isn’t sufficient to construct a shared understanding of the code. His laundry list of poor coding and documentation practices led him to the conclusion that perhaps nothing had changed about the documentation aspect of development since he started developing scientific software in the 70s.

The turnover in scientific software development projects adds to the risk of not documenting code well enough for others to comprehend it. This is because high turnover increases the likelihood that the original code author will not be available to consult with. S07 stated this as a specific reason for why she doesn’t tolerate poor documentation; her project has had a long lifespan, and she expects the graduate students working on it to come and go.

4.10 Data Management for Scientific Software

Several of our interviewees were faced with a significant data management problem due to the large amount of data used and/or created by their software. Some of the data sources and data sizes described by our interviewees were:

- A proliferation of astronomical images from observatories; approximately 80Mb of data from each data collection cycle (S03)
- Software inputs of 100 Mb and outputs of 40Gb (S16)
- Satellite images that can total over 300Gb in one month (S04)
- Images of an ocean at various depths taken every half hour; one grad student has a half a terabyte of data (S06)
- Real-time medical scans of 20-500Mbs each from three sources thirty times each second for an hour (S12)
No interviewee mentioned a particular tool they used to manage their data; some chose to do their data management without using a tool at all. S03 stated specifically that his data management was manual; he saved data in a common directory accessible by everyone in his development group and used a file naming scheme that included all the information he needed to identify a file. The main problem he encountered was that students didn’t stick with his file naming convention when they saved local copies of images on their own computers. Instead of having the long file names to rely upon, students named their copies data~1 and data~2, which made it difficult to identify which images they had copied. S04 and S06 also kept track of their data themselves.

4.11 Version Control of Scientific Software

Interviewees fell into five broad categories of version control techniques:

1. No version control (S04)
2. Some version control is done without the use of tools, but it isn’t consistently applied (S01, S02, S05)
3. Version control is done consistently without the use of tools (S06, S13)
4. Some version control is done with the use of tools, but it isn’t consistently applied (S16, possibly S03)
5. Version control is done consistently with tools (S07, S12, S14, S15, possibly S03)

Tools used among our interviewees for version control were the Revision Control System (RCS) used by S03, Concurrent Version System (CVS) used by S07, S12, and S16, and Microsoft Visual SourceSafe (VSS) used by S14 and S15.
In the first category, though S04 said that she struggled with putting all the code together, she does not use any version control system to help with this. In the second category, S02 does the version control for his software development group, but describes it as informal. S05 stated that the version control he usually comes across consists of a comment line with a version number, but this isn’t systematic.

S03, who fits under the fourth category, states that he encourages his students to use version control, but whether or not this is enforced in the group is unclear (which is why he is also included in the fifth category). Version control is important to him because not being able to return to an older version after modifying it would be a big problem for his group. S16 stated that his group used version control, but that it was sometimes abandoned when it was found to be inconvenient – the situations in which scientists found version control inconvenient are not known – and CVS was not used in small development groups, who instead shared files on a memory key.

Those who applied version control consistently, i.e. the interviewees who fall into the third and fifth groups, did so by a variety of means, including eschewing version control tools. S06 keeps a directory called “source” with subdirectories for each version of her program. S13 kept a record of the changes he made each day in such a way that he claims he could step back to a previous version. Though it isn’t clear whether or not S13 was aware of version control tools, S06 knew about CVS and consciously chose to do version control without it. The reasoning she gave was that her software does not go through many versions, and therefore she can keep track of it herself.
4.12 Code Reviews of Scientific Software

I found that, among our interviewees, code reviews were not a commonly applied technique for improving scientific software quality. Data on code reviews was not collected from most of the interviewees, but some of them – S03, S12, S14 and S15 – did code reviews.

Of the interviewees who did code reviews, some of them had difficulty acquiring another set of eyes to review the code due to the small size of their groups. S03 did all the code reviews for his students; S14 and S15 reviewed some sections of the software together, but in other cases the code was reviewed by S14 individually, even though he was also the sole developer of the code. The code reviews in S12’s group occur after a student’s development of the software is complete. This step determines the extent of redevelopment necessary to move to commercialization.

4.13 Testing Scientific Software – General

This section describes the general state of testing and general impressions of and attitudes toward testing among our interviewees. For descriptions and comments of particular types of testing, see the following section (4.14).

Most testing of scientific software among our interviewees can be described as unsystematic or ad-hoc. S01 was typical of the others in his approach to, and frustration with, testing: he said he had tested as much as he could, and he knew there were bugs, but he didn’t know how to go about finding them. S02, the head of S01’s development group, characterized the testing in his group as ad-hoc and disorganized. His group is continuously in what he terms a “run and fix” mode of operation, and they usually test the code by running the code in their development environment instead of by installing it; he considered these to be serious problems with his group’s testing.
S02’s impression is that universities in general are struggling with quality assurance. Given the context, S02’s use of the term “QA” seemed to be strongly associated with testing. However, given information they provided on specific types of testing done in their group, their testing is probably more thorough than most (based on a qualitative judgment of the variety of types of testing they perform, as discussed in section 4.14).

S03 also described testing as being an issue for his group. He told how code errors have been discovered in codes produced by his students up to the point of a first draft of a student’s thesis. His group does not use any testing methodology. S08 likewise cited testing as an issue and stated that the testing done in his group was not repeatable, indicating that it was not systematic. S13 also tested as he developed, but without a systematic strategy.

Some interviewees seemed to test in only a shallow or cursory way. S04 describes her testing as running the program with a simple set of inputs and confirming that the answers are what she expects. S08’s method of testing was similar. S07 believes that her program should be tested with as many examples as possible to prove that it works, but she said the PhD student who developed the computational portion of her application did not test it extensively; he only told her that he was convinced that it worked. S05 stated that the codes he works with will only produce sensible results if they are “bug-free”. It wasn’t clear what logic he was using here, but he didn’t seem to consider rigorous testing to be necessary. However, S05 also stated there is a significant danger in developers forgetting to validate parts of their scientific software before they use it, particularly with large codes.
There were a few interviewees who at least claimed to do systematic testing of their software. S06 described her testing as systematized, and S14’s appeared to be as well. S12 had a well-defined, iterative testing regimen for the commercialized software produced by his group. Whether the testing carried out by these interviewees was as systematic as was claimed cannot be determined from our data (especially for S06, who described her testing in less detail than S12 or S14).

### 4.14 Testing Scientific Software – Testing Types

This section presents the information on specific types of testing gathered from our interviewees. Testing types are listed in alphabetical order; a definition of each bolded term can be found in the glossary. Some of the testing types can overlap under some circumstances (for example, boundary testing and validation testing are not mutually exclusive).

- **Automated testing:** None of our interviewees discussed any automated testing, but S07 stated that her students intended to write automated tests for her software in the long term.

- **Boundary testing:** S02 and S06 discussed boundary testing of their scientific software. S02’s group built a boundary matrix for their inputs, examined the highest possible values, and tested with a value higher than the maximum for at least some of the inputs. They also tested how the program behaved when certain inputs were set to zero. For her program, which dealt with weather patterns in oceans, S06 tried theoretical extreme scenarios such as a 100-year storm (which is statistically supposed to occur only once every century).
• **Install testing:** Only one interviewee mentioned install testing, and their experience with it was unsatisfactory. S02’s group did install testing by having the students in the group try to install the program using the install instructions from the point of view of a user who is not computer savvy, such as a station operator at a nuclear power facility. Considering that the group later ran into trouble with installing their software onto user’s machines, it is questionable whether this technique was adequate.

• **Negative testing:** Negative testing was never identified by any of our interviewees by name. Negative testing was present in some interviewees’ testing regimes and notably absent in others. Negative testing done in S01 and S02’s group was defined by S01 as trying to break each other’s code by any means they can; the particular techniques they used to do this were not discussed. The only other group to discuss negative testing was S14 and S15, who stated that they found a significant number of the bugs in their software by using inputs that couldn’t possibly occur in reality; the term they used to describe this type of testing was “dumb testing”. Neither group seemed to be aware that there are documented methodologies for executing negative tests; as S14 put it, they weren’t sure how to do dumb testing because they didn’t know what “dumb” inputs a user might try.

• **Platform testing:** Several interviewees tested their software on different platforms, though as with other forms of testing, this was not always systematic. S01 and S02’s group tested their code on different commercial platforms; however, their method of doing this was haphazard. While S01 claimed to have tested their software on every
possible platform, this was clearly contradictory both with the fact that testing on every possible platform is essentially impossible and with the fact that they had not tested installation on platforms resembling those actually in use by their users.

S12’s platform testing was focused on testing with both Linux and Windows operating systems. His reason for doing this was to catch “quirks” with the C++ compilers on different platforms that could lead to different results; he thinks that testing on multiple platforms to help to produce more robust software. S14 tested the civil engineering software made by him and S15 on Windows 95, Windows 98, Windows 2000, and Windows XP. The extent of this testing is unknown. He also made a conscious decision not to test the software on Windows Vista; instead he read about the changes in Vista and determined that the help files in his software would not be accessible in Vista, but he remained confident that the rest of the software would not be adversely affected.

- **Regression Testing:** A couple of interviewees mentioned that they did regression testing, but the degree to which it was systematic varied between them. S14 performed regression testing on his bridge classification software by keeping a set of inputs for bridges that he repeated whenever a change was made to the program – how big a change was necessary before this regression suite was run is unknown. It is also unclear to what degree this regression testing was automated or manual. S16 said that his group recognized the need for regression testing, but as with other types of testing in the group, it was implemented in an ad-hoc way. S16 also stated that the same input data was used for regression tests for about a year or more. There was a concern associated with this. There could be a
tendency for code developers to tweak (or fit) the equations so that the input data continued to match the output data even in the presence of a coding error.

- **Unit Testing:** A couple of interviewees discussed their unit testing, but only in vague terms. S03 encourages his students to unit test, but whether this is enforced is unknown. S08’s testing was almost exclusively at the unit level.

- **Usability Testing:** This was one of the most common types of testing discussed by our interviewees. Because of the large amount of data we collected on the subject, usability testing information can be found in its own section (4.16).

- **Validation Testing:** Validation testing is the most common form of testing scientific software. It was the only type of testing some of our interviewees did. For more on validation testing, see the following section (4.15).

### 4.15 Testing Scientific Software – Validation Testing

Scientific software developers frequently use validation testing to assess the quality of their software; often validation testing is used to the exclusion of all other forms of testing. Each of the interviewees except for S16 discussed their validation testing procedures.

Validation testing of scientific software poses a particular challenge when assessing the results to determine whether the test has been passed or failed. This assessment is done using an oracle (see glossary for definition). There are three types of oracles. *Data-based* oracles include both real-world data and calculations done by hand that are compared with the output of the model
implemented in the scientific software to determine whether that output is acceptable.

*Benchmarks* are measures by which the output of the model implemented in the scientific software is compared relative to the outputs of other models. An oracle based on *professional judgment* relies on an expert or group of experts in the domain to evaluate the results based on their experience and knowledge to determine whether they are acceptable. These oracles are not mutually exclusive; three interviewees used oracles from more than one of these sources (see Figure 3).

<table>
<thead>
<tr>
<th>Table 5 Oracle Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-Based</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>S01-S02</td>
</tr>
<tr>
<td>S03</td>
</tr>
<tr>
<td>S04</td>
</tr>
<tr>
<td>S05</td>
</tr>
<tr>
<td>S06</td>
</tr>
<tr>
<td>S07</td>
</tr>
<tr>
<td>S08</td>
</tr>
<tr>
<td>S09</td>
</tr>
<tr>
<td>S10</td>
</tr>
<tr>
<td>S11</td>
</tr>
<tr>
<td>S12</td>
</tr>
<tr>
<td>S13</td>
</tr>
<tr>
<td>S14-15</td>
</tr>
</tbody>
</table>

All of these types of oracles carry the risk that they could be incorrect.

S01 and S02’s software was validated by using data gathered from several sources in the nuclear industry. S04 does her validation testing using a dataset she created herself. S06 uses a
combination of real data and expert judgment for validation testing. She initially tests by looking at the outputs of her program and deciding whether they seem right to her. Since her software deals with weather prediction, she can also input real weather data and check the forecast against the weather that actually occurs.

S07’s software models automata, so her means of testing is to input the data for simulated examples to ensure that the program output exhibits acceptable behavior – but her software has not yet been fully implemented or tested, and she will not be confident until many examples have been tested. Some interviewees used hand calculations to check the results of their software; these include S13, S14 and S15 – the two examples of civil engineering software development among our interviewees.

The classification of S12’s validation oracle is difficult based on the interview data. S12 initially tests the medical imaging software developed by his group using plastic models, then progresses to cadaveric materials from animals, then human cadavers, and finally to testing in the operating room. For some of these forms of data, they have some knowledge of what their imaging software ought to produce – for other forms of data, such as testing on actual patients in the operating room, the doctor’s expert judgment is used to determine whether the software is displaying an accurate image. Our data does not show how these oracle types are combined or delineated, but both data and expert judgment are used.

S03’s group uses two types of oracles to assess the quality of their image processing algorithms. One is an industry standard sharpness metric; the other is subjective judgment, or “eyeball analysis” – a visual comparison of the image processed with their algorithm versus others. As S03
characterized it, this is as simple as stating, “look at image A, look at image B; image B is better.” S05 stated that in his domain there are applications modeling behavior that one does not ever want to encounter in real life. This precludes the use of real data since there is little to none in existence and producing it is not desirable. He also commented that one is usually building on the work of others, and there are benchmarks that can be used to determine whether the results are better or worse than those produced by other methods.

Three interviewees who used commercial software to implement their models in the nuclear engineering field had to do validation testing. Each used a different type of oracle. S09 compared the output of his program to data on standard nuclear element performance. If his model produced a result within an error range, which could be “large” by his description, then it is considered correct. S10’s model was meant to be a simplification of another model. If his model produces results that were acceptably close to those of the more complex model, he considers them correct. Thus S10 is using the output of the other model as a benchmark for his simplified version. S08’s software was used to produce preliminary rough results to determine whether it was worthwhile to use the time-consuming, more accurate commercial simulator, so the commercial simulator’s results were the benchmark for his own software. S11’s model is related to nuclear meltdowns. As with models referred to by S05, this falls squarely in the category of an undesirable event for which there is little real data; gathering such data would likely be prohibitively dangerous. Due to his lack of validation data, his oracle is his own expert judgment. S11 looks at his results and determines whether or not they make sense to him.

Determining the reliability of any of these oracles is challenging. With regards to data-based oracles, one of the primary risks is that the oracle data may be wrong – the data may have been
collected or recorded improperly, or a hand calculation may be incorrect due to human error.

Even with industry data, among the most concrete oracles available, S02 stated that his group has had trouble validating their models. He described a “give and take” process between his development group and the industry data sources: if the answer given by the outputs of the scientific software does not match the expected outputs from the industry data being used as the oracle, it is possible that the industry data is at fault instead of the scientific software. S02 has encountered instances in which the industry data was incorrect as well as instances in which his model required adjustment. Therefore industry data is not entirely trustworthy, and some degree of professional judgment must be used to determine whether the data may be incorrect.

With benchmarks, a risk is that the benchmark does not provide a consistent comparison – there may be some way to improve the score on a benchmark without improving the model. S03 expressed his dissatisfaction with the industry standard sharpness metric used in his field, stating that the results can vary by 80% or more depending on the sampling of the image used by the benchmark. As with data-based oracles, there remains a necessary degree of expert judgment with regards to how much to rely upon the results of a benchmark.

As for expert judgment itself, a clear risk is that the expert may be wrong, especially in the absence of corroborating objective data, or that the expert’s judgment may not be as fine-grained as that of data-based oracles. S11 defined his model as being more vague and primitive than those of the other people interviewed with him partly due to his lack of validation data.

In addition to the challenge posed by obtaining a reliable oracle, validation testing of scientific software is burdened with the high risk of insufficiency. Due to the complex and continuous
nature of many scientific models, it is often very difficult, if not impossible, to determine where boundaries or singularities lie. Having two tests yield acceptable answers does not necessarily guarantee that points between or close to the test data will also yield acceptable answers. Therefore determining the range of data for which a scientific model is validated can be a highly challenging, sometimes insurmountable task. As with other domains, validation testing of scientific software can only decrease the risk that the software will provide an incorrect answer, but its limitations are even more pronounced when faced with the mathematics involved in scientific software computations.

Another source of potential insufficiency of the oracle is that the data used is too simplistic or not in the range of the data that the scientific software will actually be used to process. S04, who created her own oracle data, describes her dataset as “simple”. Whether this simple simulated data validates the software adequately enough to justifiably increase her confidence in the model’s applicability on complex inputs is questionable.

4.16 Usability Testing and User Documentation of Scientific Software

Usability risks were addressed by some of our interviewees through both usability testing and usability documentation. Some leaned more strongly toward either testing or documentation while others used an even mix of both. Many other interviewees did not address usability risks at all.

The main factor that correlated strongly with a concern about usability risk was the degree to which the backgrounds of the software developers differed from the software’s intended users. There are four main categories of relations between developer and user knowledge:

66
• The developers and the intended users may have very similar scientific backgrounds. This includes cases in which the intended users are the developers themselves, people in the same research group, or people in the same domain specialty (S01, S02, S03, S04, S05, S06, S07, S08, S09, S10, S11, S16)

• The users are non-specialists in the domain of the scientific software developer. This includes users who possess fundamental backgrounds in the scientific or engineering principles of the domain, but who are not specialists in the models defined in the scientific software (S07, S13, S14, S15)

• The users are scientists in a domain different from the developer’s scientific domain. The single case of this in our interviews is computer scientists developing software for use by medical doctors (S12)

• The intended users of the scientific software are not scientists. The single case of this in our interviews is the nuclear engineering development group developing software for use by station operators (S01, S02)

The interviewees’ software can fit into more than one of these categories if the software is intended for multiple user groups. The latter three groups are the ones who addressed usability risks by creating user documentation or doing usability testing. Of those interviewees with user characteristics fitting only in the first group, in which the developers and users have similar scientific backgrounds, no interviewees created user documentation or did usability testing. Scientific software developers view usability as a risk only when the users’ background knowledge is markedly unlike their own.
The ways in which usability testing was executed by our interviewees are:

- Testing by developers in the group from the point of view of intended users (S01, S02, see installation testing in section 4.14)
- Testing by users not typical of the intended user group (S14, S15)
- Testing by users representative of at least one intended user group different from the developers (S07, S12, S14, S15)

S07 had students in her course try out her software; students are one of the intended user groups. S12 has doctors test his imaging software on plastic models as well as S12 himself making multiple observations of the use of the software by doctors. S14 and S15 had engineering graduate students test their bridge classification software and its documentation by experimenting with the program and consulting help files. These students are not necessarily indicative of the eventual user group. S14 and S15 also had the opportunity to get engineers in the military with a variety of engineering backgrounds to test their software and help files. This group was typical of the users who the software is intended for.

Documentation intended for users also played a large role in addressing usability risks for some of our interviewees. S07’s scientific software was documented by a detailed tutorial. S14 and S15’s software was documented with extensive, tested help files written by S15, who was the domain expert. They included a large graphical component, including diagrams of parts of bridges drawn by S15 to help engineers who were not specialists in bridge engineering identify components of a bridge correctly.
S01 and S02 had somewhat different views of their user documentation, though both found it broadly unsatisfactory for their intended users. S01, a student in the group, said that he feels that the user documentation is sufficient guidance for the user to install the software. However, during a trial install on a real user’s computer, the users found it very difficult to install, if not impossible, because they lacked the necessary administrator privileges. S01 did allow that he thought users needed further help understanding error messages and more guidance on proper inputs. S02, the head of the development group, said they were having trouble with their user manual. He bluntly stated that the instructions for their software are unclear to the point where even experts in the field don’t understand them, which is even more problematic given that their software is intended for users who are not domain experts. He also said that their software has no electronic documentation, no help file, and no answers to frequently asked questions, though he recognized that there was a need for at least some of these documents to be created.

User documentation was not S12’s preferred approach to improve usability. He did not believe it was practical to expect a doctor to sit down and read the software manual, especially during a medical procedure, so he instead concentrated his efforts on doing usability testing and creating clear usability requirements through storyboarding to ensure that the way to use the software would be evident without needing to consult documentation. He prefers to keep his software’s user manuals to the slim size of an average video game manual. An opposite practice was implemented by S13, who focused completely on usability documentation over testing. He created help files that referenced a textbook apparently possessed by all users. This was not an unreasonable expectation given that the users work at one large company that commissioned the software to implement the models contained in the textbook.
4.17 Theory Documentation of Scientific Software

Theory documentation refers to any description of the model implemented by the software. In the case of our interviewees, theory documentation was expressed in the form of theory manuals, papers, theses, and textbooks. Theory documentation played a role as requirements documentation, code documentation, and user documentation.

Theory documentation was often the primary form of documentation used among our interviewees. S01 and S02’s software requirements documentation consisted of a few paragraphs about what the software was supposed to do; the actual documentation of much of the functional requirements was the theory manual defining the model implemented in the software. In their case, theory documentation also occasionally supplemented code documentation. S02 stated that the code documentation for his group’s software included references to supporting documentation, including the theory manual. In another group, the only software documentation required by S03 is the papers published by his students on their algorithms. S05 stated that the software groups he had worked with documented their research – which is the theory – but little else. S16 was skeptical about much of the documentation done in his group, but stated that the models implemented by the software were well understood because they were described in papers. This demonstrates that various forms of theory documentation are commonly the only, and most reliable, form of documentation for the scientific software.

Theory documentation occasionally serves as user documentation. S01 stated that if a user asks a question about why the software is behaving in a particular way, he might point out the relevant part of the theory manual as justification. In S08’s case, the documentation for the software he was using included a Master’s thesis. It was his main source of information about what the
software did and why it did it. The help files for S13’s software referenced the textbook – written by the developer – containing the models implemented in the scientific software.

4.18 Quality Factor Tradeoffs in Commercial Scientific Software

Some of our interviewees discussed their use of commercially available scientific software packages for their research (S02, S04, S08, S09, S10, S11, S13). One of the patterns I observed is the tradeoff between suitability (see glossary for definition) and usability in commercial scientific software. Sometimes this was dealt with successfully by some commercial software packages.

S04 expressed dissatisfaction with the commercial software she used. She felt that because the commercial software developers were not focused on her as a customer since her needs were so specialized and not typical in their market, the software did not include functionality that was important to her. She discussed a previous version of the software that she claimed had a very poor user interface but a higher level of functionality for advanced users than the newer release. The latest release of the software has a much improved user interface, but the streamlining resulted in the loss of functionality that she and other advanced users found useful in the previous release.

S09, S10, and S11 collectively experienced a similar tradeoff in commercial software they used, but it was handled more gracefully than in the case of S04. They stated that scientists and engineers were part of the commercial software’s target market. Nevertheless, they agreed that they spent most of their time in their research fighting with the software. They specifically cited the trade-off between usability and the suitability of the software to the tasks the scientists need it to perform. However, they knew it was not clear to them what they wanted to use the software
for. They understood that a clean user interface made it easier for them to use the software but
gave them fewer options. But the software they use implements extremely complex mathematical
equation solving techniques that they do not fully understand, so if it gave them complete
freedom, and hence less guidance, they stated that they would not know how to use the software.

S09, S10, and S11’s commercial software offered them two modes of input that somewhat eased
the tradeoff between usability and suitability. When possible, they used the graphical user
interface to input their models, because this was easier than the alternative of inputting the models
in a Matlab-like scripting language. However, when the user interface did not offer them all the
features they needed, they resorted to scripting. The graphical user interface’s emphasis on
usability and the scripting language’s emphasis on suitability allowed them to use whichever
would allow them to accomplish their tasks.

S13 described another instance of a dual modes for modeling scientific theories in commercial
scientific software. In the commercial software he used, he could create his models in one of two
modes: in a pseudocode supported by the scientific software or as a C++ module. Writing models
in C++ instead of pseudocode has the advantage of allowing greater flexibility, making it more
suitable to some scientific modeling tasks; the C++ code was also faster than pseudocode. The
pseudocode had the advantage of being easier to write for someone who wasn’t a skilled
programmer.

This is not intended to imply that all usability problems encountered by our interviewees were
caused by a tradeoff of usability for suitability. The commercial inductor design software used by
S08 contained all the functionality required to design inductors – inductor designers were part of
the software’s core user base according to S08 – but it had a very poor user interface. Inputting inductor designs was tedious and the program required the input of seemingly random non-physical parameters to return a sensible result. There is not a clear correlation in this case between poor usability and high suitability; the user interface difficulties do not seem to be related to giving users added functionality. However, even in this case, S08 stated without being directly asked that he wished the commercial software supported a scripting language.

Most of our interviewees who discussed commercial scientific software indicated that they were pushing the boundaries of what the commercial software was able to do. Because they were using this software for some kind of research either in an academic or industrial setting, they were naturally attempting to make the software take on tasks that it had never taken on before. This can result in commercial scientific software vendors having difficulty keeping their software up to date on suitability. S08 stated that the commercial software he used was perpetually about five years behind the latest inductor design theories. S09, S10, and S11 felt that their complaints and feature requests were eventually dealt with by the vendors of their commercial software as well, but by the time one new feature they requested was added to the software, they had already written their own method to work around the limitation. S13 stated that most of his customer support calls were made to determine how he could fool the software into doing things it wasn’t designed to do.

4.19 Transparency of Scientific Software

Scientific software developers sometimes find it difficult to validate the results of their own software, as described in previous sections. They are sometimes even more suspicious of the results produced by any closed source software they use. This can be due to historical issues with
the software. Possibly their lack of trust is due to a conflict of software output with their professional judgment, or it may be due to a mistrust of any software in which they cannot check the code themselves to ensure that it produces correct responses, or it may be due to a combination of these factors.

S02 was the first to bring up this concern in the interviews; the issue of transparency was later described by S13 in the focus group. S02 described an instance in which he called the developer of commercial scientific software he used to clarify a discrepancy between what the theory manual said the software did and what the software ought to be doing. He reached a very senior figure in the vendor company, who assured him that the mistake was only in the theory manual and was not reflected in the code. S02 expressed some dissatisfaction with this. Because the software was closed source, he could not verify for himself that the vendor was right. This increased his perception of the risk of trusting the results of the software.

S13 did not discuss his use of closed source scientific software, but he did discuss his perception of it in relation to his own software. In the scientific software he developed, he was careful to show every step of the calculation along the way to the engineers using the software in a completely transparent way. His reasoning for this was that even though it laid bare how his software worked, he could not expect fellow civil engineers to trust calculations that they could not see, since he would not do so himself. His reasoning for this was that a civil engineer who designs a structure could be liable if something ended up being wrong with their design. Therefore they cannot blindly trust a calculation without being able to verify it themselves.
Though accuracy was not specifically addressed as a main quality factor issue in our interviews, it emerged as a serious concern among the interviewees who participated in the focus group. S03 and S04 agreed that they found it either difficult or impossible to know how accurate the results computed by commercial scientific software were, because they could not look at the code to see how the results were calculated. Since accuracy was important to them, this had resulted in both deciding in the past to rewrite their own methods to perform the same tasks done by the closed source software. S03 in particular cited a case in which his professional judgment contradicted the results given to him by the commercial software. Since he could not verify that the code implemented the theory correctly due to it being closed source, he decided to trust his judgment and create a different implementation of his own. The lack of transparency resulted in such a significant unacceptable risk that the results would be either viewed as incorrect or not accurate enough to satisfy the scientists’ needs.

When their research or reputations in their field are on the line, transparency is a critical quality factor to scientific software users. Unfortunately this often contradicts the desire of closed-source software vendors. This creates a difficult situation for them; if they open their code up or show each step of the calculation, they risk competitors or customers copying their algorithms instead of purchasing their software. However, if their customers do not trust their product, they may decide to develop their own software to reduce the risk of trusting incorrect or inaccurate answers.

4.20 Open Source Software Used By Scientists

Despite the concerns with commercial and/or closed source software, our interviewees have not made a broad move to adopt open source software in its place. Some tools, such as those for
version control, have been adopted by some scientists; however, many replacements for closed-source scientific application software are viewed as having their own trust and user interface concerns.

S03 discussed an instance in which he was trying to use files that were unsupported by the commercial software he normally used. He turned to an alternative free program that did support the files, but he was so frustrated by the user interface that he gave up on it quickly and wrote his own file conversion utility for the commercial software instead. Since then, the commercial software has been updated to support the files he uses, and he has no intention of switching to another software package again. Though S04 had not tried any alternatives to her commercial software, she was skeptical that open source software would solve the problems she had with her commercial software. Since she has already invested in learning how to code in the language used by the commercial software to add modules to it, she is reluctant to lose her investment in time and energy to adopt a completely new software package.

S09, S10, and S11 discussed an open-source alternative to the commercial software they used; their impression was that its performance and user interface were poorer than the commercial software they used. Another experience one scientist had with open source software was a program given to him by his professor so he could put a graphical user interface on it. What he saw was a program passed from generation to generation of graduate students that was coded in an ad-hoc way with no scoping, variable passing, or cohesive design, making debugging a nightmare. His overall impression of open source software was that it isn’t reliable or elegant.
S06 was one interviewee who used open source software for her research. She used a program from Los Alamos and modified it to be appropriate for her application. S13 also used open source tools for his research, though he added the caveat that with free software, he often feels that he gets what he pays for, indicating that he is not enthusiastic about the overall quality of open source software.

4.21 Distribution of Scientific Software

Our interviewees discussed several different models of distribution. Some were closed source, some were open source; some distributed upon request, while others required some form of payment or quid-pro-quo.

Some scientific software is closed source and is not distributed for free, usually because it is being marketed to a specific industrial customer. S01 and S02’s software was intended to be marketed as a part of a toolset deployed to many nuclear reactors around the world. S12’s software is sold to hospitals. S13 never had ownership of the software he created; it was owned by the company he was developing it for.

Other software is closed source but freely distributed upon request. S07’s software is distributed through her website; she says that other software in her domain is also closed source. She believes that the reason closed source software is so prevalent in her domain is because developers want the professional credit for creating it. Another reason she gives is that software development in academia is competitive, not collaborative. The free software S08 used was closed source, but the developer emailed it upon request. S13 and S14 gave their software to the military, from whom it was distributed freely through a variety of sources.
S13 had a way of distributing his software that was unique among our sample. When people ask him for his models, he says he’ll give them away if they give him something he’s interested in as well. Other open source distributions are done on request with no quid-pro-quo; S03 says that he would give away his source code if asked, but no one ever has. S04 sometimes shares her code with other research groups when asked, as does S16’s development group.
Chapter 5

Conclusion

5.1 Comparison to Background and Related Work

Where our data characteristics overlap with those of Granados [5][6], our observations are compatible. There is clear evidence that scientists and software engineers approach software development in different ways. Our interviewees were usually members of small teams, usually did not hold computer science degrees, showed a general preference for procedural languages, and had development efforts that sometimes involved a significant degree of R&D.

With Wilson [7], our agreement is more nuanced. His emphasis on using tools was not borne out by our set of interviewees, some of whom were satisfied with their manual data management and version control. However, those who used version control and configuration management tools consistently seemed to get more value from version control than those who used such tools in an ad-hoc way. Wilson’s assertion [3] that a lack of knowledge of how to do systematic version control and testing was a major obstacle seems to agree with our data.

Segal’s work [1][12] on the characteristics of scientific software developers and the inadequacy of documentation is also compatible with our observations. The challenges cited in Segal’s case study of communication in collaborations between scientists and software developers were handled more effectively among our interviewees [11]. This may have been because none of the software developers in our case studies were trying to fit scientific software development into a waterfall paradigm.
Despite the differences between Carver et al.’s [14] data sources and our own in terms of size and breadth, we are in agreement on the motivations of scientific software developers and their characteristics and challenges. He claims that scientists only address quality factors as necessary to meet scientific objectives. There must be an additional caveat that scientists must also recognize those quality factors as risky, which does not always happen.

The concerns raised by Hatton and Roberts [24] and by Hatton [25] about the inadequacy of verification in scientific application software, and their assertion that the cases they examined probably reflected processes that were more mature than other scientists (which therefore makes it more likely that such errors would also appear in other domains), seems to be in line with our observations. While we did not collect a lot of data on accuracy issues, several interviewees viewed this as a concern.

5.2 Analysis and Future Work

Scientists develop software for several different purposes. For some purposes, such as medical imaging or calculating the possible load on a bridge, creating the software itself is a primary reason for development. In other cases, such as proving that one’s model works, the software is seen as a means to an end.

The latter purpose in particular can result in a software development effort in which corners are cut in requirements elicitation, design, implementation, deployment, and maintenance. Requirements may be undocumented unless such documentation is rendered necessary by an outside regulatory authority. Design may be ad-hoc and can result in software that is nearly impossible to fully comprehend or to maintain. Inadequate code documentation can increase risks
to maintainability and other quality factors, especially if the original code author departs the development group, as often occurs at universities. In many cases, documentation of theory played a significant role as software documentation for both developers and users. Testing is often ad-hoc. Testing other than validation testing may be ignored or only addressed in a cursory fashion, and validation testing can lack the reliability and scope necessary. All of these constitute sources of risk to varying degrees that the scientist may or may not be aware of – or, if he is aware, he may not take the risks seriously until they directly affect his ability to reach his scientific goals.

Some differences in implementation are potentially due to properties of the scientific domains themselves. The preference for procedural languages, especially Fortran and C, over object-oriented languages in general among our interviewees is due to the procedural nature of many of the problems being addressed. In cases with clearly defined objects, such as those that comprise a bridge, object-oriented languages can be very satisfactory for the scientists. Ultimately scientists should choose a language that has the features they need and which reduces the risks of added complexity. More research should be directed toward supporting design and development in procedural language, such as by adapting software design patterns and frameworks that can enhance quality factors such as maintainability, understandability, and testability.

Another such domain-dependent property is the cognitive complexity of some scientific domains. The people developing scientific software are highly educated in subject matter that often takes years to understand. This makes collaboration with software engineers even more challenging than usual. Eliciting requirements from customers is traditionally a key source of risk to software development, but the knowledge and terminology gaps between software engineers and scientists
and engineers can make collaboration between the two camps very time-consuming to ramp up and possibly not worth the trouble. However, in some cases where this effort has been made, the software engineer has managed to make a contribution. In other cases, scientists have sought out software knowledge relevant to them on their own and put real effort into evolving their software engineering skills to reduce risk without the overhead of collaboration with a software engineer.

The design lifecycle itself is influenced by domain factors. The volatility in the scope of the theory to be implemented by the software, or in the theory itself, contributes to the iterative approach to software development that I observed. The “do it twice” model I saw used reflects an attempt, sometimes unplanned, to reduce risk by separating development into a volatile first stage in which the theory, and hence requirements and design, are more volatile than in the second stage. Future research should focus on possible iterative development frameworks or methods that would manage such domain dependencies effectively, either by proving that an existing development framework or method satisfies their requirements or by creating something new.

Techniques for code reviews and testing must also keep in mind domain dependent factors. Code reviews, done by only a few of our interviewees, could potentially add value, but highly specialized skills are required to understand the code or other documents being inspected. The difficulty of testing scientific software is exacerbated by the complexities of many scientific models that rely on continuous mathematics, making them challenging to validate. Future research should address the question of how effective validation of scientific software can be done given these issues and constraints and should also address what other types of testing would be of the most value to scientists’ development goals.
Some of our interviewees produced software that in some way met their needs. Others were struggling. The ways in which they recognized and managed their risks was a key factor in the maturity of their development processes. The methods by which usability risks were addressed (when such risks existed and were identified) were broadly well thought-out and executed.

Users of scientific software had several points of concern with regard to usability and suitability. Some of their commercial scientific application software balanced these needs well, especially when the scientist was part of a valued user group. But transparency and the inability of some commercial software packages to keep up with the latest theory were largely unresolved problems, possibly due to the desire of commercial interests to keep code proprietary. Conversely, while open source scientific application software has the potential to largely solve such troubles, the perception of the usability and suitability of such software, as well as perceptions of their trustworthiness, were low among our interviewees. The scientific development community itself seems split along open source and closed source distribution models, with the rationale for keeping some scientific software closed source sometimes stemming from commercial goals.

Main themes to understand from this thesis are the following:

- Scientific software developers should do iterative development because it fits with the iterative nature of research itself;
- Scientific software developers should choose development languages that make the most sense to them conceptually and that have the mature language features they require. This may be a procedural language like Fortran or, in some cases, an object-oriented language;
• Scientific software developers should be aware of the theory, implementation, and usage risks that affect them and actively manage these risks;

• Scientific software developers should gain the software engineering knowledge they need to execute development successfully or to effectively communicate with software developers and testers on their team;

• There should be a concentrated effort on instituting systematic validation testing, and other types of testing should not be neglected;

• Correctness and transparency are quality factors of greatest concern to the scientists. There is a need for the software engineering community to provide assessment techniques to address these high priority concerns in the context of scientific software.

5.3 Use of Qualitative Data Analysis in our Research

Our research is not intended to be a statistical study of the frequencies that certain characteristics show up in the population of scientific software developers. Instead, our goal was to identify a broad spectrum of characteristics of their software development environment. I feel that with our 16 interviewees from a variety of scientific and engineering disciplines we obtained a broad base of data. Even with a small sample group, I was able to make interesting conjectures on how software development is done amongst scientists, using qualitative data analysis. These conjectures have been informally validated via publications in software engineering literature [29] [30] [31] [32].

Bringing knowledge from the software engineering domain and identifying and developing practices specifically for use by scientists is a new area of research. The first International
Workshop on Software Engineering for Computational Science and Engineering is May 2008.

Our work is providing a base for this specific software engineering work to go forward.
References


Appendix A

Letter of Information

Title: A Characterization of Users and Developers of Scientific Software

I am writing to request your participation in research aimed at determining any general patterns in the software development processes and software usage of scientists. The ultimate goal of our research is to find particular aspects of software engineering that scientific software developers are struggling or succeeding with and to determine ways software engineers or software engineering practices could be of help. I am a graduate student in the Faculty of Computing at Queen’s University. This research has been cleared by the Queen’s University General Research Ethics Board.

In this part of the research, we wish to interview users and developers of scientific software to learn about their experiences and views. To do this, we are conducting unstructured interviews involving 1-3 scientists and 1-3 interviewers at a time. Each participant engages in one interview of approximately one hour duration. A recording will be made of the interview so that notes can be taken at a later date. During the note-taking and analysis process, all record of the participant’s names will be removed from the notes, and occupations will be identified using general terms only. Data will be kept securely by the researchers and confidentiality is absolutely guaranteed.

We do not foresee risks in your participation in this research. Your participation is entirely voluntary. You are not obliged to answer any questions you find objectionable, and you are assured that no information collected will be reported to anyone who is in authority over you. You are free to withdraw from the study without reasons at any point, and you may request removal of all or part of your data.

This research may result in publications of various types, including journal articles and professional publications. Your name will not appear in any publication created as a result of this research. If necessary, a pseudonym or number will replace your name on all data that you provide to protect your identity. If the data are made available to other researchers for secondary analysis, your identity will never be disclosed.

If you have any questions about this project, or if you wish to withdraw for any reason, please contact Rebecca Sanders at [contact information] or Diane Kelly at [contact information]. For questions, concerns or complaints about the research ethics of this study, contact the Dean of the Faculty of Education, [contact information], or the Chair of the Queen’s University General Research Ethics Board, [contact information], email [contact information].

Sincerely,
Rebecca Sanders
Appendix B

Consent Form

We are required by the Queen’s General Research Ethics Board to obtain verbal consent for your participation in our study and ensure the following:

(i) you have read the letter of information and have had any questions answered to your satisfaction
(ii) you understand that you are participating in a study entitled A Characterization of Users and Developers of Scientific Software for the purpose of determining patterns in software development processes and the software usage of scientists
(iii) you understand that participation in this study involves an unstructured interview that will be recorded by a digital voice recorder
(iv) your participation is voluntary and any questions that you have now or later about the study can be directed to Rebecca Sanders or Diane Kelly;
(v) you can withdraw from the study at any time and request the removal of all or part of your data without consequences;
(vi) if you have any concerns or complaints about the research ethics of this study, you can contact the Dean of the Faculty of Education, [blank], or the chair of the General Research Ethics Board, [blank].
Appendix C
Glossary

**Automated Testing:** Automated testing uses tools to automatically run a series of test cases. The word “scripting” is sometimes used to differentiate the programming of tests from the programming of the application. Automated testing is not meant to replace testers; it is intended to support them by allowing them to do tests they could not do without automation and tests that would be very inefficient without tools.

**Boundary Testing:** Boundary testing finds bugs related to the values of variables on the boundaries of equivalence classes. Boundary bugs may show up as data input bugs or data management bugs. Boundary testing can also be used to test quality factors by determining system limits under a heavy load or in a harshly constrained environment.

**Code Review:** Code review is the examination of code, usually by someone other than the code author, for the purpose of finding implementation defects.

**Install Testing:** Install testing ensures that an application’s installer can make the application ready for use on a target system and that it can identify situations in which the application cannot be installed successfully.

**Negative Testing:** Negative testing attempts to show that an application doesn’t do what it shouldn’t do. Even if the application passes all its positive tests, it may still be possible to enter invalid inputs that result in an inelegant, data-destroying crash. One aim of negative testing is the discovery of faults that, if unchecked, could result in system crashes, data corruption, or a breach of system security.

**Oracle:** Some method to determine whether or not the output produced by a program is correct.

**Platform Testing:** Platform testing ensures that a program functions as required or expected on different hardware and software configurations. The hardware and software to be varied depends on the program being tested and can include operating systems, locales, web browsers, databases, network hardware, and anything else that defines the environment of the software being tested.

**Regression Testing:** Regression testing is the use of automatic and/or manual tests to determine whether any bugs have been injected as an unintended consequence of changes to an application.

**Suitability:** The quality of having properties that are right for a specific purpose.

**Unit Testing:** Unit testing is any testing done at the time code is written. Unit testing is most often done by the code author.

**Usability Testing:** Usability testing determines whether or not an intended user group of an application can use the application with a specified degree of ease.
**Validation Testing:** In the context of scientific software, validation testing is intended to determine how well the scientific model reflects or predicts reality.