SOFTWARE ENGINEERING FOR ENABLING SCIENTIFIC SOFTWARE DEVELOPMENT

by

DUSTIN HEATON

JEFFREY C. CARVER, COMMITTEE CHAIR
JEFF GRAY
RANDY SMITH
MARCUS BROWN
DAVID BERNHOLDT

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2015

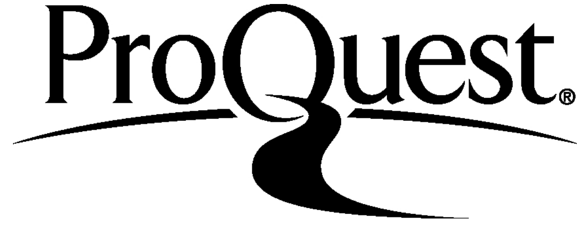
المنارة للاستشارات
www.manaraa.com

ProQuest Number: 3726076

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.



ProQuest 3726076

Published by ProQuest LLC (2015). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

المنارة للاستشارات

www.manaraa.com

Copyright Dustin Heaton 2015
ALL RIGHTS RESERVED

المنارة للاستشارات

www.manaraa.com

ABSTRACT

Scientific software is code written by scientists for the purpose of doing research. While the results of this software development have been widely published, there has been relatively little publication of the development of this software. There have been even fewer publications that look at the software engineering aspects of scientific software development and fewer still that have suggested software engineering techniques that will help scientists develop the software that is relied on for much of our modern knowledge. The software engineers who have studied the development processes of scientific software developers agree that scientists would be able to produce better software if they had the knowledge and familiarity to use specific software engineering practices. The primary focus of this dissertation is to provide that knowledge to scientific software developers in order to better enable them to produce quality software as efficiently as possible. In order to achive this focus, this dissertation has three aspects. First, this dissertation provides a literature review of the claims that have been made in the software engineering and scientific software literature culminating in a list of claims about software engineering practices. Scientific software developers can use this list to find practices they are unaware of that should prove useful to their development. Additionally, software engineers can use the list to help determine what practices need support for the scientists to be able to take advantage of them. Second, this dissertation provides a series of surveys that capture the current state of software engineering knowledge in the scientific software development community. The results of these surveys show that scientific software developers are unfamiliar with many of the practices that could help them address their

المنارة للاستشارات
www.manaraa.com

most challenging issues. Third, this dissertation provides examples that show, with support from software engineers, scientific software developers can take advantage of practices that have proven useful in traditional software engineering and increase the quality of their work without requiring an overwhelming amount of extra work.

المنارة للاستشارات
www.manaraa.com

DEDICATION

To my wife Tasha,

To my mother Susan,

To my dad Billy,

To the rest of my family and to my friends

المنارة للاستشارات
www.manaraa.com

ACKNOWLEDGMENTS

Foremost I would like to express my gratitude to my advisor, Dr. Jeffrey Carver, for the continuous support of my Ph.D. research, for his patience, motivation, enthusiasm, and supervision. His guidance helped me navigate the research and writing of this dissertation proposal.

Additionally, I would like to thank the rest of my dissertation committee: Dr. Jeff Gray, Dr. Randy Smith, Dr. Marcus Brown, and Dr. David Bernholdt.

I would also like to thank the faculty and staff from the Department of Computer Science at the University of Alabama who have trained, guided, and supported me from the beginning of my bachelor's degree studies.

I thank my current and former labmates in the Software Engineering group: Jonathan Corley, Brian Eddy, Amiangshu Bosu, Wenhua Hu, Ahmed Al-Zubidy, Elizabeth Williams, Huseyin Ergin, Amber Wagner, Debarshi Chatterji, Aziz Naanthaamornphong, and Ferosh Jacob for the discussions, the support as we approached deadlines, and for all the fun we've had when we pried ourselves away from work.

المنارة للاستشارات
www.manaraa.com

CONTENTS

ABSTRACT ii

DEDICATION iv

ACKNOWLEDGMENTS v

LIST OF TABLES x

LIST OF FIGURES xi

1 INTRODUCTION 1

1.1 Problem Statement 2

1.2 Study Rationale 5

1.3 Hypotheses 6

1.4 Methodology 7

1.5 Summary of Findings 9

1.6 Outline of Dissertation 10

2 LITERATURE REVIEW 12

2.1 Introduction 12

2.2 Background 15

2.2.1 Common Characteristics of Scientific Software Development 16

2.2.2 Variables Within Scientific Software Development 17

2.3 Methodology 18

2.3.1 Research Questions 19

للاستشارات

www.manaraa.com

2.3.2 Source Selection . . . 19
2.3.3 Study Selection . . . 20
2.3.4 Data Extraction . . . 21
2.4 Results . . . 22
2.4.1 Development Workflow . . . 23
2.4.2 Infrastructure . . . 40
2.5 Conclusion . . . 47
REFERENCES . . . 49
3 SURVEYS . . . 57
3.1 Introduction . . . 57
3.2 Survey 1 . . . 60
3.2.1 Survey Design . . . 60
3.2.2 Analysis . . . 63
3.2.3 Threats to Validity . . . 74
3.2.4 Conclusions . . . 75
3.3 Survey 2 . . . 76
3.3.1 Design . . . 76
3.3.2 Analysis . . . 78
3.3.3 Conclusion . . . 93
3.3.4 Threats to Validity . . . 96
3.4 Analysis and Conclusions Across Both Surveys . . . 97
3.4.1 Demographics . . . 97

المنارة للاستشارات
www.manaraa.com

3.4.2 Knowledge Source 97
3.4.3 Individual Practice Analysis 98
3.5 Conclusion 99
REFERENCES 99
4 CASE STUDIES 102
4.1 Introduction 102
4.1.1 Subject Selection 105
4.2 Background 106
4.3 Case Study 1: Peer Code Review 109
4.3.1 Problem and Research Objective 109
4.3.2 Background 110
4.3.3 Study Design 110
4.3.4 Results 112
4.3.5 Outcomes 113
4.4 Case Study 2: Testing 118
4.4.1 Problem and Research Objective 118
4.4.2 Background 119
4.4.3 Development of TestSci 119
4.4.4 Proof-of-Concept Example 123
4.4.5 Future Work 124
4.5 Summary and Future Work 125
REFERENCES 126

المنارة للاستشارات
www.manaraa.com

5 CONCLUSIONS AND FUTURE WORK . . . 132
5.1 Conclusion . . . 132
5.2 Contributions . . . 132
5.3 Future Work . . . 133
5.4 Publications . . . 134
REFERENCES . . . 135
APPENDICES . . . 137

المنارة للاستشارات
www.manaraa.com

LIST OF TABLES

2.1 Inclusion and exclusion criteria 20

2.2 Paper Distribution 21

2.3 Data items extracted from all the papers 22

2.4 SE practices 23

2.5 Lifecycle Model 27

2.6 Requirements 29

2.7 Design 30

2.8 Testing 34

2.9 Verification and Validation 36

2.10 Refactoring 37

2.11 Documentation 39

2.12 Issue Tracking 42

2.13 Reuse 43

2.14 Third-Party Issues 45

2.15 Version Control 46

3.1 Rating of Software Engineering Practices (Survey 1) 62

3.2 Knowledge and Familiarity Results - shaded cells indicate significance at the $\alpha <$ .05 level 69

3.3 Knowledge and Familiarity Results - shaded cells indicate significance at the $\alpha <$ .05 level 82

للاستشارات المنارة
www.manaraa.com

LIST OF FIGURES

2.1 Testing Types from Survey 31

2.2 Documentation Produced by Developers 39

3.1 Type of Employer 64

3.2 Years Developing Scientific Software 64

3.3 Knowledge Sources 65

3.4 Sufficiency of Software Engineering Knowledge (First Survey) 66

3.5 Research vs. Production (First Survey) 71

3.6 Distribution of knowledge based on development type (First Survey) 72

3.7 Summary of practices (First Survey) 73

3.8 Research vs. Production (Second Survey) 80

3.9 Reasons why developers did not use practices they felt were relevant 83

3.10 Distribution of knowledge based on development type (Second Survey) 83

3.11 Summary of practices (Second Survey) 84

3.12 Languages used in scientific software 85

4.1 Error output from TestSci 121

4.2 Code Change output from TestSci 122

المنارة للاستشارات

www.manaraa.com

Chapter 1

INTRODUCTION

Scientists and engineers use software models to replace dangerous or expensive experimentation and to conduct studies that would not be possible otherwise. The following text provides examples from three scientific domains that frequently use software models: climate science, earth science, and nuclear science. In climate science, software models allow meteorologists to predict future weather conditions and dangerous weather events. Without these models, meteorologists are limited to manually examining historical weather patterns to extrapolate predictions about future weather. This historical approach is time-intensive, which is problematic in the face of the rapid pace of changing weather conditions. Additionally, the historical approach primarily gives general predictions, which means it is likely less accurate than the results given from the models. A different problem emerges in other fields, for example many of the phenomena studied in earth science occur so slowly that it is inefficient to experiment with them physically. Software models allow earth scientists to speed up the effects of their experiments. Yet another problem emerges in the field of nuclear science: the problem of safety. It is much safer for scientists to simulate the effects of nuclear reactions than to conduct a physical experiment.

As these examples highlight, scientists and engineers are increasingly reliant on the results of software simulations to inform their decision-making process. Because of this reliance, it is vital for the software to return accurate results. While the correctness of the science behind the software is the most important factor in the accuracy of results, the correctness and quality of the software is

المنارة للاستشارات
www.manaraa.com

also extremely important. The field of software engineering provides tools and methods that help developers increase and verify software quality.

1.1 Problem Statement

Because a scientific or engineering problem must be sufficiently complex to require the development of software, developers often need advanced technical training, most frequently a PhD, in the area to understand the needs of the problem. This situation differs from a traditional software development environment such as where a high-level of domain knowledge is not as strictly required. This requirement of detailed domain knowledge frequently means that a scientific software developer lacks the software development knowledge that a "traditional" software developer would have. In turn, various aspects of software quality may be lower. Software engineering provides development methodologies, verification, validation, and testing techniques as well as version control and issue tracking tools that have the potential to increase the quality of scientific software. However, it appears that the prevalence of their use in scientific software is relatively low.

In addition to the software quality problems, scientists and engineers have a problem with productivity. According to Faulk et al., even though the speed of computers is rapidly increasing, it is becoming more difficult for scientists to actually do useful work. Faulk says that the reason for this situation is that "the dominant barriers to productivity improvement are in the software processes." In other words, the development approach that is primarily used in scientific software contains bottlenecks. Faulk also claims that these bottlenecks cannot be removed "without fundamentally changing the way scientific software is developed." [9] A major strength of software engineering is that it can increase productivity. Therefore, low productivity is another issue in which software engineering techniques can help scientific software developers.

المنارة للاستشارات
www.manaraa.com

Scientific software projects have a common set of characteristics which, according to Basili et al. [2], provide a source of knowledge that is essential to understand the claims made about the application of software engineering to scientific software projects.

First, many scientific software developers learn software development from other scientific software developers rather than via a formal software engineering education [7]. Unfortunately, the other scientific software developers also tend to lack formal software engineering training. Because scientific software developers do not have formal training, their ideas of what constitutes software engineering is limited. This lack of training means that they are likely unaware of techniques they could use that would allow them to have a much greater level of control over the quality of their code. Even when scientific software developers are familiar with certain software engineering techniques, they may not know how to properly apply them, leading them to decide that the cost of using software engineering techniques outweighs the benefits they provide.

Second, many of the software projects are not initially designed to be large, but do become large after initial trials prove successful [2]. Because the programs are not intended to be large, scientific software developers often do not take care to ensure that their code is easy to maintain. When scientific software developers have to later modify their code to add new features, these modifications require more effort than they should.

A final characteristic of scientific software is that it is generally used internally, that is either by its creator or by another member of the creator's research group [2]. Because the software is used internally, the belief is that understandability by external developers is less important. As a result, the software is often difficult to read and poorly commented. These practices lead to software that is less maintainable, which is problematic if someone new joins the team or if one of the primary developers stops working on the code for some period and then returns to it.

المنارة للاستشارات
www.manaraa.com

It is important to understand that there is not one monolithic community of scientific software developers. According to Basili et al. [2], there are three primary variables that characterize the development of software for any individual researcher or group of researchers. The first variable is *team size*. In scientific software, the size of a team is usually either a single researcher who serves as his own developer or a large group. According to Basili, the large groups tend to consist of multiple groups that may not even be co-located.

The second variable is the useful *lifetime* of the software. Software that is only expected to be executed once or twice does not require as much formal software engineering or need as much optimization as software that is going to be used multiple times, i.e. a scientific simulation or a scientific library. In this case, formal engineering is not as important because the additional work required is not justified by the lifetime of the software. Additionally, when software is only executed once or twice, the effort required to optimize its performance can easily overwhelm the performance increase this effort generates.

The final variable is the *intended users* of the software. The users can be internal, external, or both. In the case of internal users, the developers do not tend to care as much about the quality of the user interface because they will be using the software themselves. When the software is going to be used externally, the quality of the user interface is more important. Additionally, when the software is going to be modified by external developers, it must be readable and maintainable. Cases where both internal and external users are supported result in an additional layer of complication because multiple software versions must be maintained.

The previous discussions all lead into the following problem statement that my dissertation will address. *Scientific software developement has many of the same needs as traditional software development. Therefore, scientific software developers would be helped by adopting the techniques*

المنارة للاستشارات
www.manaraa.com

that have proven to be effective in traditional software engineering development. However, because there are many differences in the developers themselves, these techniques will need to be tailored to fit the scientific software context.

1.2 Study Rationale

Traditional software development focuses on fulfilling the needs of a customer. In particular, software engineers focus on a process that has been shown to lead to the creation of software that better fulfills those needs. This focus on the process has led software engineers to emphasize quality of the code itself. Scientific software, on the other hand, exists to answer scientific or engineering questions that are difficult or impossible to answer experimentally due to constraints on time, expense, or the danger of performing the experiment. Because the most important goal for scientific software developers is the creation of new scientific knowledge, the relative emphasis scientific software developers place on various software quality attributes (i.e. correctness of code, maintainability, and reliability) has been historically lower than that given by traditional software developers [8]. Furthermore, there is no guarantee that software techniques will work for scientific software development without modification. In fact, Segal, et al. suggest that software techniques would have to be tailored for use in scientific software development [13].

In software that will be used for an extended period of time, the most expensive part of development is in the maintenance stage. In some cases, this stage can take up as much as 90% of the total effort devoted to a software project. While many scientific software projects are small projects that serve as more proof-of-concept than long term software development efforts, there are also a large number of projects, such as library development or searches for a new material, that are continually developed over many years. These projects, just as with traditional software projects, will necessarily undergo a number of changes in their lifecycles. Because of this need

المنارة للاستشارات
www.manaraa.com

for continued change, the developers will be required to perform maintenance tasks on the software. Software engineers have determined that many factors contribute to the ease of maintaining software, including readability, preservation of knowledge across a team, and testing.

In addition to the long-term projects mentioned previously, scientific software developers frequently explore many similar phenomena. The development process for these similar projects would be simplified if they could more easily reuse software. Many of the same factors that contribute to maintainability also contribute to reusability: readability, preservation of knowledge, and testing.

1.3 Hypotheses

The specific areas of scientific software quality this dissertation seeks to improve are maintainability and reusability. As these two qualities greatly affect the cost of developing software in traditional software domains [3, 4], this dissertation hypothesizes that the same would prove true in the scientific software domain. In order to show that these qualities will decrease the cost to scientific software developers, this dissertation will look at the three sub-areas of readability, preservation of knowledge, and testing. Readability, preservation of knowledge, and testing are subareas of both maintainability and reusability. Improvement in each of these areas has been shown to improve both maintainability and reusability of traditional software [1, 5, 10, 14]. In traditional software engineering, code reviews have been found to have a significant impact on the readability of software and the preservation of knowledge among members of the development team [1]. In order to show that these techniques will have similar effects on scientific software development, This dissertation covers a number of hypotheses:

- The use of peer code reviews will significantly improve the readability of scientific software.

www.manaraa.com

- The use of peer code reviews will significantly increase the preservation of knowledge across a scientific software development team.
- The use of uniform coding standards across a scientific software development team will increase the readability of scientific software.
- The use of regression and integration testing in concert will provide a means for scientific software developers to show that their software is "more ."

1.4 Methodology

Because scientific software development has many of the same needs as traditional software development, this dissertation shows that software engineering techniques can be applied or modified to increase the maintainability and reusability of scientific software. In particular, this dissertation shows that peer code reviews, regression testing, and integration testing provide a noticeable increase in the quality of scientific software without requiring a large amount of additional work by scientific software developers. In order to show that these improvements can be obtained efficiently, this dissertation focuses on a number of techniques that have been shown to increase the qualities that make software maintainable and reusable. The most important criterion for both of these characteristics is "correctness." The correctness of the software is particularly important in scientific software development because the expected output of the software is often unknown. The more confident a scientific software developer can be in the correctness of his/her software implementation of the underlying algorithm, the more confident he/she can be in the correctness of that algorithm. In addition to being "correct," in order to re-use software, it must be modular enough that useful portions of the software can be used without needing to keep the parts of the software that do not apply to the new problem.

المنارة للاستشارات
www.manaraa.com

This dissertation is composed of three articles used to show that software engineering techniques can increase the maintainability and reusability of Computational Science and Engineering software as follows:

1. **Article 1 - Literature Review** to determine the views of the usefulness of software engineering techniques for scientific software development.

2. **Article 2 - Surveys** to determine the current state of software engineering knowledge and the use of software engineering practices in scientific software development.

3. **Article 3 - Case Studies**: (1) Peer code review in scientific software development and (2) Integration and Regression testing in scientific software development.

The first article describes the results of a systematic literature review conducted to understand the views of the scientific and software engineering communities on the usefulness of software engineering practices and tools for scientific software development. This literature review examined papers from both the scientific and software engineering domains and the claims made in those papers. The literature review provided a list of software engineering practices, the claims that had been made about the usefulness and effectiveness of those practices for scientific software development, and an analysis of the types of evidence used to support those claims. This last analysis showed that many of these claims have not been supported with evidence stronger than personal experience, which means that further analysis is needed.

While the literature review gave a broad overview of the claims from the literature, it did not provide as useful an overview of the practices used by the developers. Additionally, many of the claims need more extensive evaluation beyond personal experience. Therefore, in order to understand the current state of software engineering knowledge and the use of software engineering

المنارة للاستشارات
www.manaraa.com

practices in scientific software development, the second article describes the results from a series of surveys of scientific software developers. These surveys asked the developers to rate their current knowledge and use of a number of fundamental software engineering techniques. It also asked them what they saw as the major issues facing scientific software development in the future. The results showed that the developers had, in general, a lack of familiarity with the software engineering practices that support the testing and verification & validation processes important for ensuring the quality of their software. In order to address this lack of familiarity, this dissertation looks specifically at the practices of peer code review, integration testing, and regression testing.

The results of the literature review and the surveys indicated that developers faced issues specifically with maintainability and reusability of scientific software. To address these issues, the third article describes the results of informal case studies that sought to teach scientific software developer teams to utilize peer code review and provide a semi-automated tool that the teams could use to perform integration and regression testing on their software while requiring minimal extra effort on their part. The creation of this tool utilized open-source scientific software projects. The use of open source projects is important because many of these projects are standard libraries that are used in a wide range of scientific software projects. If the tool did not support these projects, then it would be of limited use to the scientific software community. These case studies showed that, with support from the software engineering community, scientific software developers could take advantage of the software engineering practices of peer code review, integration testing, and regression testing.

1.5 Summary of Findings

The results of this dissertation will prove beneficial to members of the scientific development community as they seek to maintain existing large programs or develop new programs. The

المنارة للاستشارات
www.manaraa.com

empirical evidence shown by having scientific software development teams use the software engineering techniques will help researchers evaluate the effectiveness of those techniques for their own use. The process of implementing these specific techniques in a scientific software development context will provide insight as to how much tailoring is required to make software engineering practices work in the context of scientific software development teams. The dissertation shows that, while scientific software developers recognize they would benefit from using software engineering practices, those practices that support verification & validation and testing have not been widely adopted. This lack of adoption is important because these areas have been repeatedly identified by both scientists and software engineers as some of the most difficult challenges facing scientific software development. Furthermore, this dissertation shows that scientific software developers were generally unable to evaluate their overall knowledge of software engineering as shown by their knowledge of specific software engineering practices. Finally, the dissertation shows that the software engineering practices of peer code reviews, integration testing, and regression testing are effective at addressing the issues of maintainability and readability in scientific software development.

1.6 Outline of Dissertation

This dissertation is divided into five chapters. Chapter 2 presents a literature review that examined the claims software engineers and scientific software developers have made about the usage of software practices in scientific software development. Chapter 3 presents the results of a pair of surveys that characterize the current status of the knowledge and usage of software techniques in scientific software development. Chapter 4 presents two examples showing that the software techniques of peer code review, integration testing, and regression testing are useful in the context of scientific software development teams. Chapter 5 provides an overview of the major

للاستشارات المنارة
www.manaraa.com

conclusions of my research, a plan for future work, and a listing of the publications generated by this dissertation.

المنارة للاستشارات
www.manaraa.com

Chapter 2

LITERATURE REVIEW

2.1 Introduction

Scientists and engineers often use computational modeling to replace (or augment) physical experimentation. For the remainder of this paper we will refer to the software created by these scientists and engineers as *scientific software*. The following examples help to illustrate some of the key reasons why computational models are becoming increasingly important in science and engineering domains. First, *computational models allow scientists to react to events in near real-time.* In meteorology, computational models allow scientists to adjust their forecasts based upon current conditions and analyze the potential effects of changing conditions. Without such models, meteorologists would have to extrapolate from historical data, which is time-consuming and too slow for real-time forecasts. Second, *computational models allow scientists to study phenomena that occur at a very slow pace in reality.* In climate science or geology, the slow pace of many natural phenomena make it infeasible for scientists to rely solely on empirical observations to draw conclusions. Computational models allow scientists to study these phenomena at a much more rapid pace. Third, *computational models allow scientists to study phenomena that are too precise for manual observation*. In astronomy and astrophysics, the combination of software models and advances in digital imaging systems have combined to allow scientists to discover new solar systems that are too faint for human detection. Finally, *computational models allow scientists to study phenomena that are too dangerous to study experimentally*. In astrophysics, it is much safer for

المنارة للاستشارات
www.manaraa.com

scientists to use computational models to explore the effects of various types of nuclear reactions compared with conducting physical experiments.

As these examples highlight, scientists and engineers are increasingly reliant on the results of computational modeling to inform their decision-making process. Because of this reliance, it is vital for the software to return accurate results in a timely fashion. While the correctness of the scientific and mathematical models that underlie the software is a key factor in the accuracy of results, the correctness and quality of the software that implements those models is also highly important. Additionally, the software's performance must be fast enough to provide results within the desired time window. To complicate these requirements, scientific software is typically complex, large, and long-lived. The primary factor influencing the complexity is that scientific software must conform to sophisticated mathematical models [8]. The size of the programs also increases the complexity, as scientific software can contain more than 100,000 lines of code [10, 14]. Finally, the longevity of these projects is problematic due to developer turn-over and the requirement to maintain large existing codebases while developing new code. These characteristics of scientific software development are explored forward in Section 2.2.

In the more traditional software world, software engineering researchers have developed various practices that can help teams address these factors so that the resulting software will have fewer defects and have overall higher quality. For example, *documentation* and *design patterns* help development teams manage large, complex software projects. *Version control* is useful in long-lived projects as a means to help development teams manange multiple software versions and track changes over time. Finally, *peer code reviews* support software quality and longevity, by helping teams identify faults early in the process (software quality) and by providing an avenue for knowledge transfer to reduce knowledge-loss resulting from developer turn-over (longevity).

المنارة للاستشارات
www.manaraa.com

Furthermore, software engineering practices are important for addressing productivity problems in scientific software. Even though the speed of the hardware is rapidly increasing, the additional complexity makes it more difficult for scientists to be productive developers. According to Faulk et al, the bottlenecks in the scientific development process are the primary barriers to increasing software productivity and these bottlenecks cannot be removed without a fundamental change to the scientific software development process [16].

The previous paragraphs highlighted the software quality and productivity problems that scientific software developers face. Because developers of more traditional software (i.e. business or IT) have used software engineering practices to address these problems, it is not clear why scientific software developers are not using them. Throughout the literature, various CSE researchers and software engineering researchers have drawn conclusions about the use of software engineering practices in the development of scientific software. To date, there has not been a comprehensive, systematic study of these claims and their supporting evidence. Without this systematic study, it is difficult to picture the actual effectiveness of SE practices in scientific software development. Based on our own experiences interacting with scientific software developers, we can hypothesize at the outset that the relatively low utilization of software engineering practices is the result, at least in part, of two factors: 1) the constraints of the scientific software domain and 2) the lack of formal training of most scientific software developers.

This paper has three primary contributions.

1. A list of the software engineering practices used by scientific software developers;

2. An evaluation of the effectiveness of those practices; and

3. An evaluation of the evidence used to evaluate effectiveness.

المنارة للاستشارات
www.manaraa.com

Therefore, the goal of this paper is **to analyze information reported in the literature in order to develop a list of software engineering practices researchers have found to be effective and a list of practices researchers have found to be ineffective**. In order to conduct this analysis, we performed a systematic literature review to examine the *claims* made about software engineering practices in the scientific software literature and the *claims* made in the software engineering literature about the usefulness of software engineering practices for scientific software development. In this paper, we define a **claim** as: *any argument made about the value of a software engineering practice, whether or not there is any evidence given to support the argument.* In particular, we are interested in identifying those claims that are supported by empirical evidence.

The remainder of this paper is organized as follows: Section 2.2 provides background on previous research about SE for scientific software. Section 2.3 describes the research methodology used in this systematic literature review. Section 2.4 reports the scientists' and software engineers' claims about SE for scientific software.

2.2 Background

Traditional software development focuses on the process of developing software to fulfill the needs of a customer. This focus on the process has led software engineers to emphasize quality of the code itself. Scientific software, on the other hand exists primarily to provide insight into important scientific or engineering questions that would be difficult to answer otherwise. Because the goal for scientific software developers is the creation of new scientific knowledge, the emphasis placed on software quality (i.e. correctness of code, maintainability, and reliability) has been historically lower than seen in more traditional software engineering [8]. Furthermore, even for developers who place a great deal of emphasis on software quality, it is likely that at least

المنارة للاستشارات
www.manaraa.com

some existing software engineering practices must be tailored to be effective in scientific software development [61].

The remainder of this paper focuses on the suitability of existing software engineering practices to address the issues facing scientific software developers. To provide some background, it is important to describe the scientific software community. While the scientific software community is not monolithic, Basili et al. [5] enumerated three characteristics that are common across the majority of the community. In addition to these common characteristics, Basili et al. [5] also enumerate three variables that differentiate projects within the scientific software community. The following subsections describe the common and variable aspects, respectively.

2.2.1 Common Characteristics of Scientific Software Development

These characteristics provide a backdrop that is essential to understand the claims that have been made regarding scientific software development.

1. **Source of software development knowledge** - Rather than obtaining their software development knowledge via a traditional software engineering (or computer science) education, many scientific software developers obtain their knowledge from other scientific developers (who also lack formal training). This lack of formal training often leaves scientific software developers blind to much of the field of software engineering that could provide much greater control over the quality of their code. Additionally, for those software engineering principles with which they are aware, scientific developers may be unsure of how to tailor and apply them in their particular environment. Carver et al. [7] also observed this characteristic.

2. **Unplanned increase in project size** - Rather than expending effort to initially design unproven software to be useful on a large scale, scientific software developers typically design

المنارة للاستشارات
www.manaraa.com

their software to be relatively small. Only when the software package finds success in the community does it begin to grow. As a result, later modifications become increasingly difficult and error-prone. Hinsen [24] also observed this characteristic.

3. **Typical user base** - Most scientific software (with the exception of some libraries and large commercially-available software packages - see item 3b in the next subsection) is used by its developer or members of the developer's research group. This internal use leads developers to discount usability (because they can just fix problems as they arise during use), which in turn reduces overall maintainability.

2.2.2 Variables Within Scientific Software Development

It is important to understand that there is not one monolithic community of scientific software developers. According to Basili et al., [5] there are three primary variables that help developers better understand how best to integrate software engineering practices into their specific project.

1. **Team size** – scientific software projects tend to be developed either by a single developer, who is typically also the only user, or by a large group of developers, which are often distributed.

2. **Useful lifetime of software** – There are two general types of scientific software, with a small number of projects falling between these two polls:

   (a) *Kleenex software* – intended to be used only once or twice, therefore good software engineering practices are less important.

   (b) *Community or library software* - intended to be used multiple times, often outside of

المنارة للاستشارات
www.manaraa.com

the developer group, therefore requires better software engineering practices to help ensure its correctness and performance.

3. **Intended users of the software –**

   (a) *Internal* – software engineering practices are less common because the developers care less about the maintainability of the software or the usability of the interfaces. Maintainability and usability matter less in this case for two reasons. First, the software is not usually planned to be used for an extended period of time, therefore less effort will be spent maintaining the software. Second, the software will be used by the people who developed it, so the interfaces will be used by people who already understand them.

   (b) *External* – software engineering practices are more common because the readability and maintainability of the code is more important as well as the usability of the user interfaces. Software intended for external users, on the other hand, is frequently expected to be used long term. The software will also be used by people who aren't already familiar with the interfaces, so the interfaces should be intuitive.

   (c) *Both* – results in an additional layer of complexity because teams must maintain multiple versions of the software (e.g. an internal development version and a stable release version).

2.3 Methodology

The following subsections describe the steps of the Systematic Literature Review (SLR) process we followed [38].

المنارة للاستشارات
www.manaraa.com

2.3.1 Research Questions

There have been many claims made about how scientists develop software. But as of yet, there have been no systematic reviews of the literature from both scientific software development and software engineering to collect and validate those claims. Therefore, the main purpose of this review is to survey the literature from both disciplines to answer two questions:

1. *What claims have researchers made about the usage of software engineering practices in the development of scientific software?*

2. *What empirical evidence exists to validate these claims?*

2.3.2 Source Selection

In order to gain as much coverage of the software engineering and scientific software domains, we searched the following five databases:

- ACM Digital Library,
- IEEE eXplore,
- ScienceDirect,
- SIAM Publications Online, and
- Google Scholar.

Our initial search string, *"scientific software development,"* returned an overwhelming number of results, many of which were irrelevant. The revised search strings, *"scientific software development" AND "software engineering,"* resulted in a more manageable 349 papers. Additionally, in order to be sure we had found all relevant papers, we repeated the search using each of the terms

www.manaraa.com

Table 2.1: Inclusion and exclusion criteria

Inclusion Criteria	**Exclusion Criteria**
Paper must be in the scientific software domain	Studies not in English
Paper must focus on the development of scientific software	Preliminary Conference Versions of included journal papers
The development section must mention SE topics	Study does not make claims about SE topics
	Study is a book chapter, introduction, or index

in Table 2.4 in place of "software engineering." These additional search strings resulted in a total of 718 papers. We conducted this search throughout May 2015.

2.3.3 Study Selection

We used the following steps to reduce those 718 papers down to the most relevant set to include in the review.

1. *De-duplication:* Remove any duplicates from the returned papers;
2. *Title-based exclusion:* Use the title to eliminate any papers clearly not related to the research focus;
3. *Abstract-based exclusion:* Use the abstract and keywords to exclude papers not related to the research focus; and
4. *Full text-based exclusion:* Read the remaining papers and eliminate any that do not fulfill the criteria described in Table 2.1.

The de-duplication and title-based exclusion steps eliminated 459 papers, leaving 259. The abstracts did not contain sufficient information to eliminate any additional papers. Finally, the full text review allowed us to eliminate 32 papers that did not give enough detail about the development of the software and 161 that did not make any claims about software engineering practices.

المنارة للاستشارات
www.manaraa.com

Table 2.2: Paper Distribution

Source	Count
Computing in Science and Engineering	15
ICSE Workshop on Software Engineering for Computational Science and Engineering	10
IEEE Software	6
IEEE International Conference on Software Engineering	4
ACM-IEEE International Symposium on Empirical Software Engineering and Measurement	3
ACM Conference on Computer Supported Cooperative Work and Social Computing	3
International Conference on Software Testing, Verification, and Validation	2
SIAM Journal on Scientific Computing	2
Empirical Software Engineering	1
IEEE Power Engineering Society Winter Meeting	1
International Journal of High Performance Computing Applications	1
CTWatch Quarterly	1
IEEE International Conference on e-Science	1
Advances in Computers	1
Computer	1
IEEE International Geoscience and Remote Sensing Symposium	1
European Conference on Software Architechture Workshops	1
ACM Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery	1
IEEE International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering	1
ACM International Conference on Supporting Group Work	1
SIAM Journal on Matrix Analysis and Applications	1
HPC-GECO/CompFrame Workshop	1
Workshop on Algorithm Engineering and Experiments	1
SIAM Journal on Discrete Mathematics	1
IEEE International Conference on Electro/Information Technology	1

Table 2.2 shows the distribution of publication venues for the 66 papers that made it to the data extraction step. One paper was published in a non-peer reviewed source, Advances in Computers. This paper was included as it provided a significant amount of information.

2.3.4 Data Extraction

Table 2.3 shows the items contained in the data extraction form we used to ensure consistent and accurate gathering of information from each paper. During the data extraction process, the

المنارة للاستشارات
www.manaraa.com

Table 2.3: Data items extracted from all the papers

Data items	Description
Identifier	Unique Identifier for the paper
Bibliographic	Author, year, title, source
Domain	The domain of the project the paper is based on
Claims	A list of the claims the paper made about various SE techniques
Evidence for Claims	A list of the evidence the paper provided to justify their claims about each technique

first author performed the primary extraction for the review while the second extracted data from a random sample of 5% of the papers. We then compared the data extracted by each reviewer for consistency. We found that the data extracted from the samples by the second author was consistent with the data extracted by the first author. This process is consistent with the process followed in previous systematic reviews [20, 27, 28, 37, 68].

2.4 Results

In our review of the literature, we used the definitions provided by the IEEE Standard Computer Dictionary [1] to categorize the claims about the effectiveness of software engineering practices in scientific software into 11 practices. We then divided these practices into two groups: (1) those that are primarily part of the software development workflow, and (2) those that are part of the infrastructure that supports software development. Table 2.4 lists the 11 practices and the two larger groupings. The remainder of this section describes the claims about each of these 11 practices in more detail. Throughout the discussion, we emphasize the claims with bold-faced text and provide additional discussion to substantiate the claim. While the standard practice in systematic literature reviews is to provide separate answers for each research question, in this review we believed it made more sense to answer them together so that each claim would be

المنارة للاستشارات
www.manaraa.com

Table 2.4: SE practices

Development Workflow	Design Issues Lifecycle Model Documentation Refactoring Requirements Testing Verification and Validation
Infrastructure	Issue Tracking Reuse Third-Party Issues Version Control

presented along with the evidence that supports it. Additionally, while any particular claim may be positive or negative, the claims are worded so that all evidence supports them.

2.4.1 Development Workflow

Many of the claims focus on elements of the software development workflow, which usually includes requirements, design, implementation, testing, refactoring, and documentation. The following subsections address each of these practices. The claims made in the following subsections are summarized in Tables 2.7-2.11. First the claim is presented and then the papers that made the claim are categorized under the type of evidence they gave to support that claim. The first category is NS, or No Specific evidence given. The second is PE, or Personal Experience. The third category is I/S, or Interviews and Surveys. The final category is CS, or Case Study. The papers categorized into the Case Study category did not neccessarily perform a formal case study, but they did observe the practice in use outside of their own personal experience.

www.manaraa.com

2.4.1.1 Lifecycle Model

Our literature survey identified sixteen studies that contained claims about the use of lifecycle models by scientific software developers. Table 2.5 summarizes the five claims that are described in detail below.

LM1: Scientific software developers generally do not use a formal software development methodology. We identified nine studies that made this claim [2, 10, 14, 34, 36, 44, 46, 62]. Instead of using a formal develoment methodology, scientists develop their software as follows:

1. The developer forms a basic idea of what is needed and begins coding.
2. The developer informally evaluates the software through questions like "does this software do what I want?" and "Can it be usefully extended?"
3. The developer either modifies or extends the code as appropriate until the answer to the first question above is "yes," and the answer to the second is "no."
4. The developer "tests" the software by asking, "Is the output broadly what I expect?"

When the answer to step 4 is "yes," the developer considers the project complete. This approach is only successful when the developer has a thorough understanding of the domain and what is required to solve the problem, the developer is either the only user or part of the community that will be using the product, and the software is meant to answer a "particular problem for a particular group at a particular point in time." [62]

LM2: The development methodology used by scientific software developers is similar to the agile development methodology. A series of studies have suggested that scientific projects are well-suited for agile development methodologies [3, 10, 14, 32, 44, 46, 64]. When scientific

المنارة للاستشارات
www.manaraa.com

projects are investigating new science, they are not able to determine all of the requirements in advance. Therefore, they cannot effectively use plan-driven approaches. Instead, the development teams need a methodology that allows them to experiment with different solutions as the requirements are discovered. This methodology would have to include many of the characteristics of the agile development methodologies developed by software engineers [10]. Scientific software developers support the observation that they generally use an agile development approach because they do not know the requirements ahead of time [14].

Another scientific software development team suggests that the theoretical appropriateness of agile-like approaches give benefits in the real world. The team adopted ideas from the agile methodologies to successfully address the specific needs of their project. Their team was spread across multiple labs and projects, which resulted in a need for "good communication across the team, rapid development and delivery, and project management to coordinate development and manage dependencies." The need for communication is addressed by daily stand-up meetings that allow members to help each other through issues and discuss new ideas. The needs for rapid development and project management are met by iteration planning meetings, where they created plans for short development cycles [36].

LM3: Paired programming provides an effective method for dealing with complex software development requirements and an effective avenue for knowledge transfer. Paired programming ensures that all developers on their team are able to take part in design and implementation decisions. Additionally, paired programming provides a convenient avenue for knowledge transfer among the team, both of software development and subject matter knowledge. Paired programming is also very valuable for the development of complex software functions, particularly when one developer is incorporating parts of the other developer's software into their own

المنارة للاستشارات
www.manaraa.com

code [36]. Conversely, some scientists also say that paired programming is not natural for scientific software developers, and so it is not useful in all cases [21, 34].

LM4: Other software development approaches can also be useful in the right setting. Scientific software developers viewed three more development practices as useful, however these were not as broadly studied:

1. **Feature-driven development** was successful for one team [34],
2. **Test-driven development** reduces the number of errors introduced into the code for multiple teams [2, 46, 57], and
3. **Iterative/incremental development** allows developers to get around the need for an up-front requirement document that is required in a waterfall type model [6, 62]. For example, a team had previously attempted to use a linear development methodology, but found that it was completely unsuited for their needs and resulted in an unsuccessful first phase. They then adopted an iterative development method for the second phase and found it was successful [52].

LM5: Existing software development methodologies will need to be tailored to the specific context of any given scientific software development team. In the more traditional business/IT domain for teams fewer than half of the teams use a published methodology. In fact, many teams tailor the methodology to fit their particular project [61]. For example, agile methodologies should be the best fit for scientific software developers as these methodologies value: "response to change over following a plan," "individuals and interactions over processes and plans," and "working software over comprehensive documentation." However, agile methodologies alone would not work in every case. In one project, because of existing interfaces, there were certain requirement

المنارة للاستشارات
www.manaraa.com

Table 2.5: Lifecycle Model

Claim	NS	PE	I/S	CS
LM1: Scientific software developers do not generally use a formal software development methodology	[2]	[36, 62, 64]		[10, 14, 34, 44]
LM2: The development methodology used by scientific software developers is similar to the agile development methodology		[3, 32, 36]	[64]	[6, 10, 14, 44, 46]
LM3: Paired programming provides an effective practice for dealing with complex software development requirements and an effective avenue for knowledge transfer		[21, 36]		[34]
LM4: Scientific software developers viewed feature-driven development, test-driven development, and iterative development as effective	[2, 52]	[62]		[34, 46, 57]
LM5: Existing software development methodologies will need to be tailored to the specific context of any given scientific software development team		[61]		

specifications that had to be met. This portion of the project is better handled by a traditional development method. In order to address the discrepancy, it was effective to utilize a method from Boehm and Turner to blend agile and traditional methodologies in order to minimize the risk in the development process. This blend incorporated particular agile elements into the project development. As an example, the project had a long time-scale which meant that knowledge of the instrument, software and science had to be preserved. To address this need for preservation of knowledge, the development team utilized pair programming where a software developer was paired with scientist so that the developer learns some of the scientific background of a project and the scientist becomes familiar with the software [61].

المنارة للاستشارات
www.manaraa.com

2.4.1.2 *Requirements*

Our survey of the literature identified five studies that contained claims about the use of requirements by scientific software developers. We group the detailed list of claims into three over-arching claims in the following discussion. Table 2.6 summarizes the three claims that are described in detail below.

RQ1: Scientific software developers often do not produce a proper requirements specifications. Multiple studies have made this claim [34, 40, 59–62]. In one particular set of interviews of scientific developers, none created a requirements document. Even in cases where the sponsor mandated production of a requirements document, the developers created it when the software was almost finished. The most information an interviewee had at the start of development was a vision statement from a customer [59]. In another example, scientists developed using an iterative approach which allowed the requirements to emerge over time rather than being articulated *a priori*. Therefore, the lack of understanding of the need for up-front requirements, led to late document delivery and increased time pressure on the project [61].

RQ2: When scientific software developers do produce requirements, they generally focus on high-level requirements. We found two studies that made this claim [40, 62] Traditionally, because the high-level requirements are part of the scientist's domain knowledge, they tend to assume that the task of translating these high-level functional requirements into lower-level requirements would be trivial for software engineers. However, because software engineers may not have the requisite scientific background to perform this decomposition, projects end up being more expensive than necessary [62].

RQ3: When scientists produce high-level requirements, they rely on developers to pri-

المنارة للاستشارات
www.manaraa.com

Table 2.6: Requirements

Claim	NS	PE	I/S	CS
RQ1: Scientific software developers often do not produce proper requirements specifications.		[62]	[59, 60]	[34, 40, 61]
RQ2: When scientific software developers produce requirements, they generally focus on high-level requirements.		[62]		[40]
RQ3: When scientists produce high-level requirements they rely on developers to prioritize them.			[60]	

oritize them. In one case we found [60], the lack of scientific background made it difficult for the software developers to properly prioritize requirements and effectively develop the software [60]. To rectify this problem, a scientific representative had to be assigned in a later stage to prioritize requirements for the developers [60].

2.4.1.3 Design Issues

Our survey of the literature identified four studies that contained claims about the use of software design by scientific software developers [60]. We group the detailed list of claims into three over-arching claims in the following discussion. Table 2.7, summarizes the three claims that are described in detail below.

DI1: In general, scientific software designers do not treat design as a distinct step in the development process. In the first study related to design issues, the authors interviewed twelve scientific software developers. Only two of the interviewees actually used a separate software design step. Most of the interviewees had backgrounds similar to those of their users. This common background led the scientific software developers to assume that a design that suits the use of the designer will also suit the needs of the user. There were only two scientists, a civil engineer and a medical software developer, that could not assume that the user would not be similar enough to the designer for the same design to suit their needs. It was these two that performed a distinct design

المنارة للاستشارات
www.manaraa.com

Table 2.7: Design

Claim	NS	PE	I/S	CS
DI1: In general, scientific software designers do not treat design as a distinct step in the development process.			[60]	
DI2: Scientific software developers see redesign as a waste of time that risks breaking the "science" of a program.			[60]	
DI3: Object Oriented Design helps produce useful software.		[17, 19, 51]		

step. A civil engineer took advantage of the object-oriented design philosophy and a medical software developer utilized a software architecture that had been used for previous medical software projects.

DI2:Scientific software developers see redesign as a waste of time that risks breaking the "science" of a program. These scientists said they added modules to "behemoth" or "monster" programs that were the culmination of years of work by multiple researchers. Some of the interviewees would only consider redesign if runtime was a critical factor for the software's success that was not being met. In general, the scientists did not view design as an important practice due to either not seeing it as providing an advantage or their development consisting mostly of expanding existing software projects that they are reluctant to change [60].

DI3:Object Oriented Design helps produce useful software. The other three studies [17, 19, 51] dealt with projects that sought to provide modular functionality. Each of the authors found that the use of a programming language that allowed them to utilize an Object-Oriented design paradigm was neccessary for their project to be successful.

2.4.1.4 Testing

Our survey of the literature identified eighteen studies that contained claims about the use of testing by scientific software developers. We group the detailed list of claims into four over-arching

المنارة للاستشارات
www.manaraa.com

claims in the following discussion. Table 2.8, summarizes the four claims that are described in detail below.

T1: The effectiveness of the testing practices currently used by scientific software developers is limited. Ten studies made this claim [2, 13, 14, 30, 43, 45, 47, 54, 55, 62]. One of these studies was a survey, the results of which are summarized in Figure 2.1.

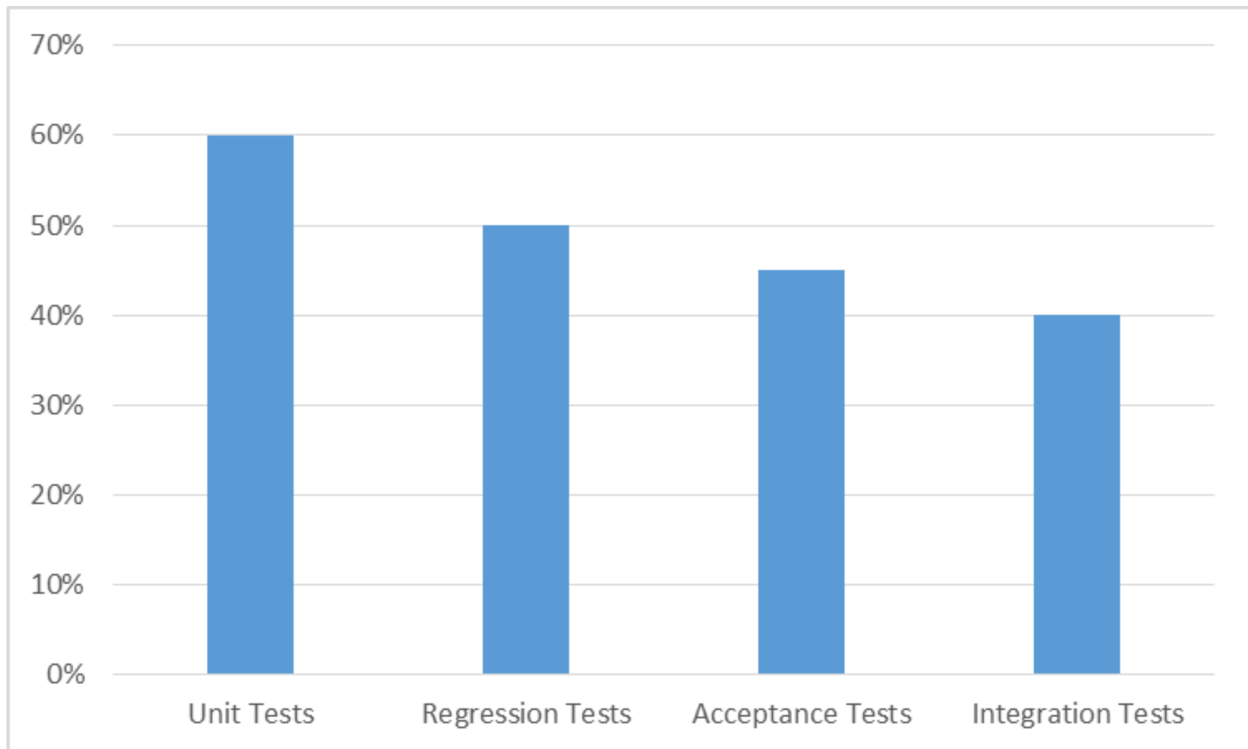


Figure 2.1: Testing Types from Survey

The authors also asked respondents to give the reasons that they performed certain types of tests. They received the following responses (in order of the number of responses):

- Correctness of software;
- Known results or ‘reliable’ programs to compare against exist;
- Easiest or least effort required for these tests;

المنارة للاستشارات
www.manaraa.com

- User acceptance;
- Not testing software is 'stupid';
- Considered to be best practices;
- Familiarity with methods; and
- Avoid costly maintenance later.

A few respondents also gave reasons that they did not perform testing: 1) "lack of management support," 2) "applications are not large or complex enough to warrant certain types of testing," and 3) "it is usually clear whether the software is working as intended." Even these people who gave reasons not to perform testing utilized two or more types of testing [47].

T2: Scientific software developers benefit from using a wide range of testing practices from software engineering. Twelve studies made this claim [2, 12, 13, 30, 33, 41, 43, 48, 49, 52, 55, 56]. One method of addressing the problem in T1 is to use test-driven development to keep bugs such as these from remaining in their code in addition to doing a regular, automated build in order to test their code on a regular basis rather than waiting until project is completed [2, 13, 49, 55]. Additionally, according to the Los Alamos National Laboratory (LANL) Accelerated Strategic Computing Initiative ASCI Software Engineering Requirements, regression testing and integration testing are both essential elements of software project management [13, 52]. In fact, many scientists who successfully test their code are actually using integration testing already, but they just think of it as using the scientific method. For example, every time a model is changed, the scientists treat it as a new experiment and test it, using the previous results as a control [14, 43].

T3: The testing practices that scientific software developers do utilize are often exe-

المنارة للاستشارات
www.manaraa.com

cuted poorly. When testing is inconsistent, the tests are not repeatable. A potential pitfall is the possibility that scientists use testing to show that the theory is correct, rather than using testing to identify where the software does not work properly. This choice could be due to the code being tightly coupled to the theory in the scientist's mind, not existing as an entity of its own. Testing requires comparison to an oracle. The problem is that when the oracle and test results do not match, the scientist does not know whether the problem lies with the theory, the theory's implementation, the input, or if the oracle itself is flawed [12, 33]. This last case can occur even when an oracle is available from measurements of a physical experiment–the measurements can be incorrect or incomplete. Furthermore, even with a perfect oracle, the fact that two tests yield the correct answer does not mean that similar, but not the same, inputs will yield the correct answer [59].

T4: Testing is much more complicated for scientific development than traditional software development since the correct results are frequently unknown. An additional difficulty is that testing is much more complicated for scientific software developers due to the fact that experimental validation may be impossible. This lack of experimental validation means that a scientist may not even have an expected answer [14, 29, 32, 62]. Much of this difficulty largely seems to stem from developers testing their code after development is mostly finished, forcing the developers to test the software as a whole instead of breaking it into realistically testable pieces [2, 13]. One study proposed a practice to test scientific programs without relying on experimental validation. In particular, they used metamorphic testing, assertion testing, and generated less rigorous testing oracles using machine learning [29].

2.4.1.5 Verification and Validation

Our survey of the literature identified eleven studies that contained claims about the use of verification and validation by scientific software developers. It is worth noting that scientific

المنارة للاستشارات
www.manaraa.com

Table 2.8: Testing

Claim	NS	PE	I/S	CS
T1: The effectiveness of the testing practices currently used by scientific software developers is limited.	[2]	[13, 43]	[47]	[14, 30, 45, 54, 55, 62]
T2: Scientific software developers would benefit from using a wide range of testing practices from software engineering.	[2, 52]	[12, 13, 41, 43, 49]		[30, 33, 48, 55, 56]
T3: The testing practices that scientific software developers do utilize are often executed poorly.		[12, 33]	[59]	
T4: Testing is much more complicated for scientific development than traditional software development since the correct results of a piece of software are frequently not known.	[2]	[13, 29, 32, 62]		[14]

software developers and software engineers do not neccessarily use the same definition of verification and validation. A common definition of verification in scientific software development is "the process of determining if a computational model obtained by discretizing a mathematical model of a physical event and the code implementing the computational model can be used to represent the mathematical model of the event with sufficient accuracy" [4]. A similar definition for validation is "the process of determining if a mathematical model of a physical event represents the actual physical event with sufficient accuracy" [4]. While these are slightly different from the standard software engineering definitions of verification as the evaluation of how well a product corresponds with its specifications and validation as the evaluation of how well a product meets its goals, they are similar enough that we will use the software engineering definition in the remainder

المنارة للاستشارات
www.manaraa.com

of this section. We group the detailed list of claims into four over-arching claims in the following discussion. Table 2.9, summarizes the four claims that are described in detail below.

VV1: The lack of suitable test oracles or comparable software makes validating scientific software difficult. There are two primary issues raised by the studies that have made this claim. First, there is a lack of suitable test oracles to use for scientific software development [14, 22, 32, 55, 56]. There are rarely other pieces of software that are both relevant to the problem a developer is working on and already have exact answers the developer could compare against. Because this software rarely exists, scientific software developers find it hard to compare the results from their software with the results from other pieces of scientific software [22, 32, 53, 55, 56]. To address this lack of external information, some developers have attempted to perform verification by monitoring variables that change in a known manner, but these variables are not the ones that scientists are usually concerned with, so the usefulness of this monitoring is limited [53]. Additionally, the models that are implemented in scientific software are usually extremely complex, and the value of a model to scientists does not necessarily depend on how exactly it matches reality [53].

VV2: There are many ways that defects can enter software. First, the science behind the code could be wrong. Second, the translation from the scientific model to an implementable algorithm could be wrong. Finally, the translation from algorithm to code could be wrong [7, 9, 55].

VV3: Scientists frequently suspect that any problems in the results of their software result from scientific theory. One study found that when validation testing fails, scientists tend to look more closely at the science rather than the code. This finding indicates a problem because the lack of attention allows errors in code to slip through unnoticed [31].

VV4: Experimental validation is frequently impractical because scientists lack the

المنارة للاستشارات
www.manaraa.com

Table 2.9: Verification and Validation

Claim	NS	PE	I/S	CS
VV1: The lack of suitable test oracles or comparable software makes validating scientific software difficult.		[22, 32, 55, 56]		[14, 53]
VV2: There are many ways that defects can enter software.		[55]		[7, 9]
VV3: Scientists frequently suspect that any problems in the results of their software result from scientific theory.		[31]		
VV4: Experimental validation is frequently impractical because scientists lack the information they would prefer to use to validate the software.			[25, 63]	[10, 14]

information they would prefer to use to validate the software. We found this claim in four studies [10, 14, 25, 63]. There are two primary reasons given for this claim. First, many scientists believe that useful validation would have to consist of comparing the results from their software to the results gained from a physical experiment or observation [14]. As was mentioned in the introduction, the cost or danger of performing these physical experiments is often the reason why scientists build software models in the first place, so the physical experiments will not be done before promising software models have been created. Second, expermental validation is frequently impractical since it is usually difficult or impossible to know what the correct result for a piece of software will be until the software is run [10, 25, 63]. In some cases, scientific software developers treat validation studies as research projects or theses in and of themselves due to the challenge in performing them. In these cases, scientists do not find that it is feasible to fully validate every piece of their software [25].

2.4.1.6 Refactoring

Our survey of the literature identified five studies that contained claims about the use of refactoring by scientific software developers. We group the detailed list of claims into two over-

المنارة للاستشارات
www.manaraa.com

Table 2.10: Refactoring

Claim	NS	PE	I/S	CS
RF1: Refactoring is a useful practice to increase software quality.	[2]	[15]		[11, 39]
RF2: Refactoring is not always possible.				[14]

arching claims in the following discussion. Table 2.10, summarizes the two claims that are described in detail below.

RF1: Refactoring is a useful practice to increase software quality. Four of the five studies found that refactoring was a useful practice. In particular software refactoring:

- is a useful practice for improving performance [11, 15],
- has proven to be a highly valuable practice for the bioinformatics domain [11],
- is also a powerful practice for maintaining and improving the quality of code [11, 39], and
- is particularly useful in conjunction with the automated refactoring tools of IDEs such as Eclipse [2].

RF2: Refactoring is not always possible. On the other hand, refactoring is almost impossible when bit-wise comparison (i.e. a practice in which a model is run for a shortened period of time and then the variables are compared with other runs of the same length) is used to verify code, because that practice would only work if changes did not alter the bit values of any of these variables [14].

2.4.1.7 Documentation

Our survey of the literature identified nine studies that contained claims about the use of documentation by scientific software developers. We group the detailed list of claims into three

المنارة للاستشارات
www.manaraa.com

over-arching claims in the following discussion. Table 2.11 summarizes the three claims that are described in detail below.

D1: Documentation is a necessary enabler of software quality. Formal documents are important when a project is given to a team that is not the original development team [21, 24, 32, 50]. In fact, examinations of the ASCI program at LANL and Lawrence Livermore National Laboratory (LLNL) suggest that documentation is one of the practices that is essential for scientific software developers to adopt in order to guarantee quality [9, 52]. One benefit is that documentation enables communication between team members as well as providing references for papers, grant authoring, and grant reporting. Documentation is especially vital if scientific software developers wish to use their earlier software as a basis for developing more advanced software [24, 39, 54].

D2: Documentation is becoming more frequently used. In a more recent study, Nguyen-Hoan et al. [47] conclude, based on the result of a survey with 60 respondents, that documentation is more widely produced than is indicated in these early studies. The responses to their summary are given in Figure 2.2. Nguyen-Hoan et al. also gave the three most common arguments in favor of producing comments as well as the four most common reasons for not producing comments. The comments in favor were: 1) “For users of the software,” 2) “For future maintenance purposes,” and 3) “Documentation is integral to software.” The arguments against documentation were: 1) “Limited due to time and effort required,” 2) “Effort not worth it due to small user base,” 3) “requirements constantly changing or not specified up front,” and 4) “Software should be or is ‘intuitive,’ ‘easy to understand,’ or ‘doesn’t need a full description’ [47].”

D3: Documentation requires a significant investment of work. Not all of the claims about documentation were positive. The effort required to create documentation leads some developers to conclude that scientific software developers should be careful about how much docu-

المنارة للاستشارات
www.manaraa.com

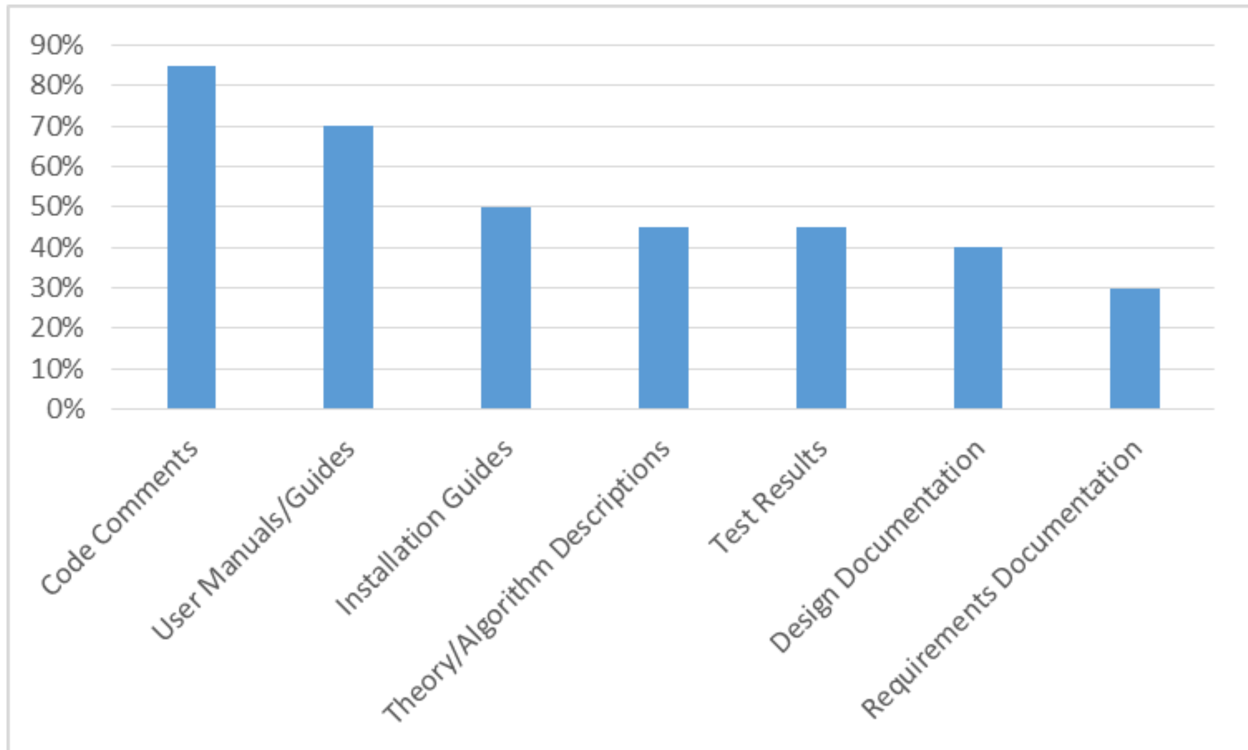


Figure 2.2: Documentation Produced by Developers

Table 2.11: Documentation

Claim	NS	PE	I/S	CS
D1: Documentation is a necessary enabler of software quality.	[52]	[21, 24, 32]	[50]	[9, 39, 54]
D2: Documentation is becoming more frequently used.			[47]	
D3: Documentation requires a significant investment of work.		[21]	[50]	[54]

mentation they create [21, 50]. Furthermore, if the documentation is done to satisfy an external requirement it may not benefit the project team [21, 50]. However, one study did find an alternative to performing a separate documentation task and utilized an automatic documentation generator that creates documentation from comments in the project's source files [54].

2.4.1.8 Summary of Development Workflow Claims

In general, the development workflow claims suggest that each practice would be useful, but there are difficulties that keep scientific software developers from adopting them in their current

المنارة للاستشارات
www.manaraa.com

forms. Seventeen claims were supported by multiple types of evidence with seven supported by only one type of evidence. The most common type of evidence for these claims that were supported by only one type was Interviews and Surveys, which accounted for three of the claims. Only four studies had support from every type of study. The claim that had the most support was *T2: "Scientific software developers would benefit from using a wide range of testing practices from software engineering."* Notably, every claim that was supported by a paper that did not provide evidence was also supported by papers that provided one of the other types of study.

We identified two claims with conflicting evidence. First, one author's personal experience was that paired programming is valuable for the development of complex software while the case study from another author showed that it is not useful for scientific software development. This issue needs to be further examined as there are two primary possible explanations. The first is that the usefulness of paired programming depends on the context of the development team. The second potential explanation is that the teams involved in the case study were not well-trained in utilizing paired programming or they lacked the knowledge to utilize this practice effectively. Additionally, while they viewed documentation as necessary, two studies made the claim that the amount of work required to create documentation means that scientific developers should be careful about how much documentation they make.

2.4.2 Infrastructure

The practices in this category all serve to support various aspects of the software development lifecycle. Each practice is addressed in its own section and the claims are summarized in Tables 2.12-2.15. The tables use the same notation as described in Section 2.4.1

المنارة للاستشارات
www.manaraa.com

2.4.2.1 *Issue Tracking*

Our survey of the literature identified two studies that contained claims about the use of issue tracking by scientific software developers. We group the detailed list of claims into two over-arching claims in the following discussion. Table 2.12 summarizes the two claims that are described in detail below.

IT1: Issue tracking greatly eases communication between members of a development team. Issue tracking is a useful practice for communicating information about discovered bugs and needed functionality between developers. Issue tracking software allows this information to be stored and communicated instantly between all members of a development team. When additional remote groups are added to existing development teams, an issue tracking system is required to formally record bugs and new requirements as well as to create a trail of the completed activity [2]. Issue-tracking software also made a list of the ten most important software engineering practices for scientific software developers to use [21]. There are four primary reasons to use issue tracking software rather than informally tracking issues:

- Issues can be made visible to the entire team;
- The ability to prioritize issues is frequently provided;
- Many systems provide the ability to track dependencies between issues; and
- The history of issues is searchable for future reference [21].

IT2: Issue tracking helps insure that no two groups in a development team are working on the same problem. The dependency-tracking feature of issue tracking software also allows a large deliverable to be broken into a set of smaller features that it is dependent upon. This set

www.manaraa.com

Table 2.12: Issue Tracking

Claim	NS	PE	I/S	CS
IT1: Issue Tracking greatly eases communication between members of a development team.	[2]	[21]		
IT2: Issue Tracking helps insure that no two groups in a development team are working on the same problem.		[21]		

can then be distributed among various groups so that no two groups are working on the same feature [21].

2.4.2.2 Reuse

Our survey of the literature identified eight studies that contained claims about the use of reuse by scientific software developers. We group the detailed list of claims into two over-arching claims in the following discussion. Table 2.13 summarizes the two claims that are described in detail below.

RU1: Software must be properly designed to be reusable. The primary reasons to reuse a piece of software are to save time and money as well as to ensure reliability. The following qualities are needed for a reusable component: self-contained, able to be combined with other components with minimal side effects, formal mathematical basis, confidence that the component performs its defined purpose satisfactorily, understandable, verifiable, encapsulation, simple interface, flexibility, easily modified, general, programming language independent, and portable [24]. The most reused types of artifacts are source code, scripts, algorithms, and practices [19, 26, 35, 44, 58, 66].

RU2: There are many reasons not to reuse or produce reusable software. The primary barriers to the reuse of software are that available software does not meet requirements closely enough and that the software was difficult to understand or poorly documented. There are also

المنارة للاستشارات
www.manaraa.com

Table 2.13: Reuse

Claim	NS	PE	I/S	CS
RU1: Software must be properly designed to be reusable.	[44]	[19, 24, 35, 58]	[66]	[26]
RU2: There are many reasons not to reuse or produce reusable software.		[58, 65]	[66]	[26]

many reasons for not producing reusable code: the additional expense of developing for reuse, the software release policies of their organizations, concerns over intellectual property rights, and the absence of a common distribution mechanism [26, 58, 65, 66].

2.4.2.3 Third-Party Issues

Our survey of the literature identified nine studies that contained claims about the use of third party software by scientific software developers. We group the detailed list of claims into four over-arching claims in the following discussion. Table 2.14 summarizes the three claims that are described in detail below.

TPI1: Third party software may cease being supported before scientific software projects are finished. The long lives of scientific software projects make it likely that any particular technology will cease being supported before the project is finished. These long lives have led to many instances of technologies that promised improved productivity only to cease being supported and no longer be available [5, 42]. Due to this history of failed usage of third-party technologies, scientific software developers tend to prefer to either develop the software they need themselves or to use open-source software.

TPI2: Scientific software developers are not convinced that reusing existing frameworks will save effort in their development. Some scientific software developers believe it takes more effort to fit their work into a framework than they would save by using the frameworks [5, 42]. Additionally, a significant barrier to the use of existing frameworks is that they cannot be integrated

المنارة للاستشارات
www.manaraa.com

incrementally into an existing code. Scientists tend to mitigate risk by having multiple technologies co-existing within a piece of software until one is chosen, but this practice cannot be followed easily with many frameworks [5].

TPI3: Open-source is especially useful to scientific software developers. Five studies found that open source had promising features to help scientific software developers utilize third party products. First, the scientists do not have to devote their effort to developing the software. Additionally, if the original developers of the software cease to support it, scientific software developers have access to the source code and can maintain it themselves [3, 10, 18, 26, 35, 42, 44, 67]. In one project, in order to limit the risks, a developer was assigned to thoroughly test any code before it was integrated into the main project. The project itself was set up so that commitments to the sponsor are not endangered by the absence of an expected piece of third party software [10].

TPI4: Open-sourcing software can be seen as giving up a competitive advantage. One study, however, found that the competitive nature of the scientific community can lead developers to not produce open-source software in the first place. The study concluded that this is a problem that can only be directly addressed by encouraging the community to communicate more closely and recognize that shared development will allow science to advance at a greater rate [67].

2.4.2.4 Version Control

Our survey of the literature identified ten studies that contained claims about the use of version control by scientific software developers. We group the detailed list of claims into two over-arching claims in the following discussion. Table 2.15 summarizes the two claims that are described in detail below.

VC1: Version control software is necessary for research groups with more than one developer. This claim was made by eight studies [2, 5, 6, 14, 16, 25, 31, 54]. Version control tools

www.manaraa.com

Table 2.14: Third-Party Issues

Claim	NS	PE	I/S	CS
TPI1: Third Party Software may cease being supported before scientific software projects are finished.		[42]	[5]	
TPI2: Scientific software developers are not convinced that reusing existing frameworks will save effort in their development.		[42]	[5]	
TPI3: Open-source software is especially useful to scientific software developers.		[3, 35, 67]		[10, 18, 26, 42, 44]
TPI4: Open-sourcing can be seen as giving up competitive advantage.		[67]		

are needed to keep up with changes to software that can accumulate extremely rapidly. Version control tools allow a developer to track each new version of a piece of code that is created and identify changes between versions. Versions of software that are used to publish results, support major decisions, or undergo extreme testing are the most important to track [31]. In addition, the developer needs to maintain a complete copy of any software used to produce important results. An example of why the need to keep a copy of the software is important is a case where a researcher tried to reproduce results that she had produced the previous year. Because the researcher did not have access to the old versions of the data she needed, she had to spend a considerable amount of time reproducing the input data. In one case, even though she had the data, the results were significantly different from the previous run. In this case, the executable for the first test had been built using different compiler options [31]. In addition to utilizing version control systems, it is useful to have a formal process to approve code that is to be checked into the repository. This process would ensure that a piece of code passes all relevant test cases instead of relying on individual developers to perform these tests [25].

VC2: Distributed version control is particularly useful for scientific software devel-

المنارة للاستشارات
www.manaraa.com

Table 2.15: Version Control

Claim	NS	PE	I/S	CS
VC1: Version control software is necessary for research groups with more than one developer.	[2]	[16, 31]	[5, 25]	[6, 14, 54]
VC2: Distributed Version Control is particularly useful for scientific software development.	[2]			[14, 54]

opment. There are two major types of version control systems: centralized and distributed. In a centralized system, a single server stores the master copy of the entire project; meaning that if the server goes down or the network becomes unavailable, no one can submit work on the project. Also, if the server's data is lost, the entire project is lost as well. An alternative is to use a distributed version control system instead. In this implementation, each user has a full copy of the entire project on their machine which is updated to match other copies as connections to the other nodes in the network are available. The primary problem with distributed version control systems is that they are complicated to manage, requiring a strategy for sharing modifications and synchronizing local copies [23]. One instance of distributed version control is "The Abinit forge," a custom version control system built on Bazaar–a distributed version control system and an ssh-server. Each Abinit developer has a Bazaar repository that stores their branches of their software, providing fast data access and somewhat optimized usage of disk space. A daily script makes all contributions to a project available through a password-protected website which allows the developer to share his work with others and organize collaborative developments involving remote workplaces [54]. Other tools in common use are: Revision Control System and Concurrent Version System, the latter of which can be utilized from within the Eclipse IDE to ease the overhead required by adopting the tool [2, 14].

المنارة للاستشارات
www.manaraa.com

2.4.2.5 *Summary of Infrastructure Claims*

In general, the infrastructure claims suggest that each of the practices covered under each practice would be useful, but there are difficulties that keep scientific software developers from adopting them in their current forms. Despite this positivity, the adoption of these practices is not particularly wide-spread in scientific software development. This lack of adoption suggests that this area needs the support of software engineers seeking to aid scientific software development.

The claims related to infrastructure have not been investigated as deeply as those related to the development workflow. In fact, there were only ten major infrastructure claims. Eight of the claims were supported by multiple types of evidence, while two were supported by only one type of evidence. Both of these were supported by Personal Experience. Three claims had support from every type of study. Two claims tied for having the highest level of support: *TPI3: "Open-source software is especially useful to scientific software developers"* and *VC1: "Version control software is necessary for research groups with more than one developer."* Once again, every claim that was supported by a paper that did not provide evidence was also supported by papers that provided one of the other types of study. It is interesting to note that all evidence indicated that the infrastructure practices are effective. Even so, the general lack of strong support suggests that the entire area of infrastructure support for scientific software development is an open research area for software engineers seeking to aid scientific software developers in their pursuit of knowledge.

2.5 Conclusion

This paper looked at the literature on development in computational science from both the scientific software and software engineering domains in order to answer the question of what claims are made about the usage of software engineering practices by scientific software devel-

المنارة للاستشارات
www.manaraa.com

opers. In order to answer this question, the paper looked at the claims made by both scientific software developers and software engineers. The claims show that scientific software developers believe that software engineering practices could increase their ability to develop quality software and software engineers agree that scientific software developers adopting software engineering practices would allow them to produce higher quality software. Scientific software developers and software engineers agree that computational science has much work left to do in order for the quality of computational science software to reach the level of quality that characterizes traditional software. In particular, every piece of evidence from these papers indicated that the infrastructure practices are effective when they are used. Despite this, there was a general lack of strong supporting evidence, which suggests that the entire area of infrastructure support for scientific software development is an important research area for software engineers seeking to assist scientific software developers.

Even so, these claims show that there has been much work done in this area already. Specific practices that have been adopted strongly by scientific software developers are issue tracking and version control. The authors who addressed these practices also saw them as two of the most important practices. Interestingly, scientific software developers have, to a large extent, unconciously adopted a software engineering development practice. While many scientific software developers do not know how to formally implement the agile development approach, their normal development methodology closely approximates it. The agile approach fits well with their inability to know all of the requirements of the physical systems their software is attempting to model. The iterative nature of the agile approach also allows the software models to evolve more easily than a traditional waterfall model would.

The practices that scientific software developers view as important, but have not widely

www.manaraa.com

adopted, are verification and validation and testing. While they see these practices as extremely useful and important, the current status of each of them is extremely difficult in the scientific software domain. In particular, it is difficult to validate scientific software by comparing it to real-world data because the scientific software is attempting to investigate areas for which real-world experiments are not feasible to perform. Additionally, scientific software developers do not know how to apply many of the testing practices that have been developed in traditional software engineering to their own software development. Because of this lack of knowledge, software engineers need to work with scientific software developers to train the scientific software developers in testing and verification and validation practices. Software engineers also need to tailor the existing practices to better fit the needs of the scientific software developers.

المنارة للاستشارات
www.manaraa.com

REFERENCES

[1] "IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries," *IEEE Std 610*, pp. 1–217, Jan. 1991.

[2] K. Ackroyd, S. Kinder, G. Mant, M. Miller, C. Ramsdale, and P. Stephenson, "Scientific software development at a research facility," *IEEE Software*, vol. 25, no. 4, pp. 44–51, 2008.

[3] S. Ahalt, B. Minsker, M. Tiemann, L. Band, M. Palmer, R. Idaszak, C. Lenhardt, and M. Whitton, "Water science software institute: An open source engagement process," in *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering*, 2013, pp. 40–47.

[4] I. Babuska and J. T. Oden, "Verification and validation in computational engineering and science: Basic concepts," *Computer Methods in Applied Mechanics and Engineering*, vol. 193, no. 36–48, pp. 4057–4066, Sep. 2004.

[5] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz, "Understanding the high-performance-computing community: A software engineer's perspective," *IEEE Software*, vol. 25, no. 4, pp. 29–36, Jul. 2008.

[6] R. Betz and R. Walker, "Streamlining development of a multimillion-line computational chemistry code," *Computing in Science and Engineering*, vol. 16, no. 3, pp. 10–17, May 2014.

[7] J. C. Carver, "Report from the second international workshop on software engineering for computational science and engineering," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 14–19, 2009.

[8] ——, "SE-CSE 2008: The first international workshop on software engineering for computational science and engineering," in *Companion Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 1071–1072.

[9] J. C. Carver, L. M. Hochstein, R. P. Kendall, T. Nakamura, M. V. Zelkowitz, V. R. Basili, and D. E. Post, "Observations about software development for high end computing," *CTWatch Quarterly*, vol. 2, no. 4A, pp. 33–37, 2006.

المنارة للاستشارات
www.manaraa.com

[10] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 550–559.

[11] C. Crabtree, A. Koru, C. Seaman, and H. Erdogmus, "An empirical characterization of scientific software development projects according to the boehm and turner model: A progress report," in *Proceedings of the ICSE Workshop on Software Engineering for Computational Science and Engineering*, May 2009, pp. 22–27.

[12] P. Dubois, "Testing scientific programs," *Computing in Science and Engineering*, vol. 14, no. 4, pp. 69–73, Jul. 2012.

[13] ——, "Maintaining correctness in scientific programs," *Computing in Science and Engineering*, vol. 7, no. 3, pp. 80–85, 2005.

[14] S. M. Easterbrook and T. C. Johns, "Engineering the software for understanding climate change," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 65–74, 2009.

[15] W. R. Elwasif, B. R. Norris, B. A. Allan, and R. C. Armstrong, "Bocca: A development environment for HPC components," in *Proceedings of the 2007 Symposium on Component and Framework Technology in High-performance and Scientific Computing*, 2007, pp. 21–30.

[16] S. Faulk, E. Loh, M. L. Vanter, S. Squires, and L. G. Votta, "Scientific computing's productivity gridlock: How software engineering can help," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 30–39, 2009.

[17] A. George and J. Liu, "An object-oriented approach to the design of a user interface for a sparse matrix package," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 4, pp. 953–969, 1999.

[18] I. Gorton, Y. Liu, C. Lansing, T. Elsethagen, and K. Kleese van Dam, "Build less code deliver more science: An experience report on composing scientific environments using component-based and commodity software platforms," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, 2013, pp. 159–168.

[19] L. Hall, C. Hung, C. Hwang, A. Oyake, and J. Yin, "COTS-based oo-component approach for software inter-operability and reuse (software systems engineering methodology)," in *Proccedings of the IEEE Aerospace Conference*, vol. 6, 2001, pp. 2871–2878.

[20] J. E. Hannay, D. I. K. Sjoberg, and T. Dyba, "A systematic review of theory use in software engineering experiments," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 87–107, Feb. 2007.

المنارة للاستشارات
www.manaraa.com

[21] M. A. Heroux and J. M. Willenbring, "Barely sufficient software engineering: 10 practices to improve your CSE software," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009, pp. 15–21.

[22] D. Higdon, M. Kennedy, J. C. Cavendish, J. A. Cafeo, and R. D. Ryne, "Combining field data and computer simulations for calibration and prediction," *SIAM Journal on Scientific Computing*, vol. 26, no. 2, pp. 448–466, 2004.

[23] K. Hinsen, K. Laufer, and G. K. Thiruvathukal, "Essential tools: Version control systems," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 84–91, 2009.

[24] K. Hinsen, "Software development for reproducible research," *Computing in Science and Engineering*, vol. 15, no. 4, pp. 60–63, 2013.

[25] L. Hochstein and V. R. Basili, "The ASC-Alliance projects: A case study of large-scale parallel scientific code development," *Computer*, vol. 41, no. 3, pp. 50–58, 2008.

[26] X. Huang, X. Ding, C. P. Lee, T. Lu, and N. Gu, "Meanings and boundaries of scientific software sharing," in *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, 2013, pp. 423–434.

[27] E. Insfran and A. Fernandez, "A systematic review of usability evaluation in web development," in *Proceedings of the 2008 International Workshops on Web Information Systems Engineering*, 2008, pp. 81–91.

[28] M. Jorgensen and M. Shepperd, "A systematic review of software development cost estimation studies," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 33–53, Jan. 2007.

[29] U. Kanewala and J. Bieman, "Techniques for testing scientific programs without an oracle," in *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering*, May 2013, pp. 48–57.

[30] K. Karhu, T. Repo, O. Taipale, and K. Smolander, "Empirical observations on software testing automation," in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, 2009, pp. 201–209.

[31] D. Kelly, D. Hook, and R. Sanders, "Five recommended practices for computational scientists who write software," *Computing in Science and Engineering*, vol. 11, no. 5, pp. 48–53, 2009.

[32] D. Kelly, S. Smith, and N. Meng, "Software engineering for scientists," *Computing in Science and Engineering*, vol. 13, no. 5, pp. 7–11, Sep. 2011.

المنارة للاستشارات
www.manaraa.com

[33] D. Kelly, S. Thorsteinson, and D. Hook, "Scientific software testing: Analysis with four dimensions," *IEEE Software*, vol. 28, no. 3, pp. 84–90, May 2011.

[34] R. Kendall, J. Carver, D. Fisher, D. Henderson, A. Mark, and D. Post, "Development of a weather forcasting code: A case study," *IEEE Software*, vol. 25, no. 4, pp. 59–65, 2008.

[35] D. I. Ketcheson, K. Mandli, A. J. Ahmadia, A. Alghamdi, M. Q. de Luna, M. Parsani, M. G. Knepley, and M. Emmett, "Pyclaw: Accessible, extensible, scalable tools for wave propagation problems," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C210–C231, 2012.

[36] S. Killcoyne and J. Boyle, "Managing chaos: Lessons learned developing software in the life sciences," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 20–29, Nov. 2009.

[37] B. Kitchenham, E. Mendes, and G. H. Travassos, "Cross versus within-company cost estimation studies: A systematic review," *IEEE Transactions on Software Engineering*, vol. 33, no. 5, pp. 316–329, May 2007.

[38] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University and Durham University Joint Report, Tech. Rep. EBSE 2007-001, 2007.

[39] Y. Li, "Reengineering a scientific software and lessons learned," in *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering*, 2011, pp. 41–45.

[40] Y. Li, N. Narayan, J. Helming, and M. Koegel, "A domain specific requirements model for scientific computing," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 848–851.

[41] C. Martínez, L. Moura, D. Panario, and B. Stevens, "Locating errors using elas, covering arrays, and adaptive testing algorithms," *SIAM Journal on Discrete Mathematics*, vol. 23, no. 4, pp. 1776–1799, 2010.

[42] D. Matthews, G. Wilson, and S. Easterbrook, "Configuration management for large-scale scientific computing at the uk met office," *Computing in Science and Engineering*, vol. 10, no. 6, pp. 56–64, 2008.

[43] M. Miic, M. Tomaevic, and I. Bethune, "Automated multi-platform testing and code coverage analysis of the CP2K application," in *Seventh International Conference on Software Testing, Verification and Validation*, Mar. 2014, pp. 95–98.

المنارة للاستشارات
www.manaraa.com

[44] J. Y. Monteith, J. D. McGregor, and J. E. Ingram, "Scientific research software ecosystems," in *Proceedings of the 2014 European Conference on Software Architecture Workshops*, 2014, pp. 91–96.

[45] C. Morris and J. Segal, "Some challenges facing scientific software developers: The case of molecular biology," in *Fifth IEEE International Conference on e-Science*, 2009, pp. 216–222.

[46] A. Nanthaamornphong, K. Morris, D. Rouson, and H. Michelsen, "A case study: Agile development in the community laser-induced incandescence modeling environment (CLiiME)," in *5th International Workshop on Software Engineering for Computational Science and Engineering*, May 2013, pp. 9–18.

[47] L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana, "A survey of scientific software development," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 12:1–12:10.

[48] R. Nori, N. Karodiya, and H. Reza, "Portability testing of scientific computing software systems," in *Proceedings of the 2013 IEEE International Conference on Electro/Information Technology*, May 2013, pp. 1–8.

[49] C. Omar, J. Aldrich, and R. C. Gerkin, "Collaborative infrastructure for test-driven scientific model validation," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 524–527.

[50] A. Pawlik, J. Segal, and M. Petre, "Documentation practices in scientific software development," in *Proceedings of the 5th International Workshop on Cooperative and Human Aspects of Software Engineering*, Jun. 2012, pp. 113–119.

[51] F. Pluquet, S. Langerman, A. Marot, and R. Wuyts, "Implementing partial persistence in object-oriented languages," in *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments*, 2008, pp. 37–48.

[52] D. E. Post and R. P. Kendall, "Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: Lessons learned from ASCI," *International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 399–416, 2004.

[53] D. E. Post, R. P. Kendall, and R. F. Lucas, "The opportunities, challenges, and risks of high performance computing in computational science and engineering," in *Quality Software Development*, ser. Advances in Computers, M. V. Zelkowitz, Ed. Elsevier, 2006, vol. 66, pp. 239 – 301.

المنارة للاستشارات
www.manaraa.com

[54] Y. Pouillon, J. Beuken, T. Deutsch, M. Torrent, and X. Gonze, "Organizing software growth and distributed development: The case of Abinit," *Computing in Science and Engineering*, vol. 12, no. 1, pp. 62–69, (Jan-Feb) 2011 2011.

[55] H. Remmel, B. Paech, C. Engwer, and P. Bastian, "Design and rationale of a quality assurance process for a scientific framework," in *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering*, May 2013, pp. 58–67.

[56] H. Remmel, B. Paech, P. Bastian, and C. Engwer, "System testing a scientific framework using a regression-test environment," *Computing in Science and Engineering*, vol. 14, no. 2, pp. 38–45, Mar. 2012.

[57] M. Rilee and T. Clune, "Towards test driven development for computational science with pfunit," in *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, Nov. 2014, pp. 20–27.

[58] S. Samadi, N. Almaeh, R. Wolfe, S. Olding, and D. Isaac, "Strategies for enabling software reuse within the earth science community," in *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium*, vol. 3, Sept 2004, pp. 2196–2199 vol.3.

[59] R. Sanders and D. Kelly, "Dealing with risk in scientific software development," *IEEE Software*, vol. 25, no. 4, pp. 21–28, Jul. 2008.

[60] J. Segal, "Software development cultures and cooperation problems: A field study of the early stages of development of software for a scientific community," *Computer Supported Cooperative Work*, vol. 18, no. 5–6, pp. 581–606, 2009.

[61] ——, "When software engineers met research scientists: A case study," *Empirical Software Engineering*, vol. 10, no. 4, pp. 517–536, 2005.

[62] ——, "Some challenges facing software engineers developing software for scientists," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009, pp. 9–14.

[63] F. Shull, J. Carver, L. Hochstein, and V. Basili, "Empiricalrical study design in the area of high-performance computing (HPC)," in *Proceedings of the 2005 International Symposium on Emperical Software Engineering*, 2005, pp. 17–18.

[64] M. T. Sletholt, J. Hannay, D. Pfahl, H. C. Benestad, and H. P. Langtangen, "A literature review of agile practices and their effects in scientific software development," in *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering*, 2011, pp. 1–9.

المنارة للاستشارات
www.manaraa.com

[65] E. H. Trainer, C. Chaihirunkarn, A. Kalyanasundaram, and J. D. Herbsleb, "Community code engagements: Summer of code & hackathons for community building in scientific software," in *Proceedings of the 18th International Conference on Supporting Group Work*, 2014, pp. 111–121.

[66] ——, "From personal tool to community resource: What's the extra work and who will do it?" in *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, 2015, pp. 417–430.

[67] M. J. Turk, "Scaling a code in the human dimension," in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, 2013, pp. 69:1–69:7.

[68] B. J. Williams and J. C. Carver, "Characterizing software architecture changes: A systematic review," *Information and Software Technology*, vol. 52, no. 1, pp. 31–51, 2010.

المنارة للاستشارات
www.manaraa.com

Chapter 3

SURVEYS

3.1 Introduction

Scientific software is software developed and used by scientists or engineers to support research in a number of important fields, e.g. climate modeling, weather forecasting, high-energy physics, and cancer research. The development and use of scientific software is increasingly enabling important discoveries by replacing dangerous or expensive physical experimentation and aiding in the processing of very large data sets. Because the output of scientific software is a key factor in many critical decisions, it is of utmost importance for that software to be of high quality and produce correct results. The higher the quality of the software, the more confidence researchers and decision-makers can have in the results.

Before a scientific or engineering problem requires the development of software to support its investigation, it must be of sufficient complexity. Therefore, the scientific developers often need advanced technical training in the domain, likely a PhD, to even understand the problem. A typical software engineer will lack this level of knowledge, making it difficult for him or her to be an effective developer. As a result, scientists and engineers, who generally lack formal software engineering education, must also serve as the main developers of scientific software.

Traditional software is software developed for use in the business/IT world. Software engineering researchers and practitioners have developed a number of practices to reduce development effort and increase software quality for traditional software. Unfortunately, scientific software de-

المنارة للاستشارات
www.manaraa.com

velopers often do not adopt these practices [12]. While the level of adoption is increasing, there is still room for additional improvement [13]. It is likely that the low adoption rate of software engineering practices affects the quality of the resulting scientific software.

Various researchers have conducted studies to understand the factors that may influence the low adoption rate of software engineering practices within the scientific software community. In comparing the characteristics of traditional software with those of scientific software, researchers have noted some similarities and differences. Some of the differences suggest the need for tailoring software engineering practices for effective use in the scientific software domain [14, 21]. These differences include:

- scientific software developers learn how to develop software from other scientific software developers who also lack formal software engineering training [1];
- many large scientific software packages were not initially designed to be large, but rather grew as a result of success [1];
- scientific software is primarily used by its developer or its developers' research group rather than by external users [1];
- scientific software requirements gathering and discovery is difficult because the goal of the software is often to explore unknown domains to increase understanding rather than to solve a known, tractable problem [4–6, 19];
- verification and validation are difficult because often the expected result is not known or cannot be known *a priori* and there are multiple potential sources for defects [4, 5, 15, 17]; and

المنارة للاستشارات
www.manaraa.com

- the scientific or engineering outcomes are viewed as being more important than choosing the most appropriate software engineering practices [4, 5, 18].

These characteristics suggest that scientific software developers could benefit from using software engineering practices. They also suggest that scientific software developers are not taking advantage of, or are not aware of, these potentially helpful practices.

An important step prior to addressing the need for improved software engineering for scientific software is to understand the current state of software engineering knowledge and use in the scientific software community. This paper describes the results of two surveys we conducted to gather this information. To that end, the main contributions of this paper are:

- a detailed examination of the current state of software engineering in the scientific software community,
- identification and analysis of the most important problems in scientific software development, and
- an analysis of the current state of knowledge and use of the software engineering practices that could address those problems.

We published preliminary results from Survey 1 in *Computing in Science and Engineering* [2]. In addition, we presented preliminary results from and Survey 2 at the *First Workshop on Maintainable Software Practices in e-Science* [9]. This paper extends the prior work in two ways. First, it adds further analysis and new data from each survey. Second, it provides a detailed analysis of the results across both surveys.

The remainder of this paper is organized as follows. Section 3.2 describes the design and results of the Survey 1. Section 3.3 describes the design and results of Survey 2, including the

المنارة للاستشارات
www.manaraa.com

changes made to address weakness of Survey 1. Finally, Section 3.4 describes the findings across both surveys and draws some conclusions.

3.2 Survey 1

The goal of Survey 1 was to gather information from scientific software developers about:

1. Their perception of the level of software engineering knowledge possessed by themselves, their team, and the scientific software community as a whole;
2. How they acquired their software engineering knowledge; and
3. Whether they were familiar with and/or using various standard software engineering practices.

3.2.1 Survey Design

To address these goals, we designed a 4-part survey. The complete survey is available at `http://carver.cs.ua.edu/Data/Journals/CSE-Surveys/survey1.pdf`.

First, to obtain an overall picture of software engineering knowledge, the respondents answered the following questions on a 3-point scale (*Not Sufficient*, *Mostly Sufficient*, *Fully Sufficient*) and provided an explanation:

1. "Do you think **your** current knowledge and skills about software development and software engineering are sufficient to effectively meet your project's objectives?"
2. "Do you think **your team members'** current knowledge and skills about software development and software engineering are sufficient to effectively meet your project's objectives?"
3. "In general, do you think the current knowledge and skills about software development and software engineering in the **scientific software community** are sufficient to effectively advance scientific software?"

المنارة للاستشارات
www.manaraa.com

Second, our previous work, as well as supporting evidence from other researchers, led us to believe that most scientific software developers lack formal software engineering training [4, 5, 12, 13]. To validate this belief, the survey asked the respondents to rank order the following sources relative to their frequency of use in obtaining software engineering knowledge:

- reading books,
- attending training courses,
- co-workers,
- learned own my own.

Third, the bulk of the survey focused on specific software engineering practices commonly used in traditional software development. The respondents provided the information necessary to complete Table 3.1, where the value for each cell was chosen from a 5-point scale (*1-none* to *5-very high*)[1]. We chose these practices because our interactions with members of the scientific software community led us to believe they would be potentially useful for scientific software developers and at least some were already in use. The survey did not provide a definition for each practice because we believed that the terms were self-evident – a threat to validity discussed in more detail in Section 3.2.3. The design of Survey 2 also addresses this threat.

Fourth, the survey gathered the following demographic information about the respondents: type of institution in which they work (government lab, university, private company, other), number of years of scientific software development, fraction of scientific software development devoted to producing software for use in their own research vs. software intended for use by others, and the level and field of their educational degrees.

[1] Note that the survey did not present this exact table, rather it presented the questions separately. We use the table in the paper for compactness.

المنارة للاستشارات
www.manaraa.com

Table 3.1: Rating of Software Engineering Practices (Survey 1)

	Personal Use	Personal Familiarity	Team Use	Team Familiarity	Relevance to My Work
Software Lifecycles					
Documentation					
Requirements					
Basic Design					
Intermediate Design					
Verification & Validation					
Unit Testing					
Integration Testing					
Acceptance Testing					
Regression Testing					
Version Control/ Change Management					
Issue/Bug Tracking					
Test-Driven Development					
Structured Refactoring					
Code Review					
Agile Methods					

المنارة للاستشارات
www.manaraa.com

3.2.2 Analysis

Rather than organizing the results around the exact outline of the survey from Section 3.2.1, the following subsections make use of data from various parts of the survey to provide a more complete picture of the state of software engineering in the scientific software community.

3.2.2.1 Demographics

Our goal in this survey was to target as large a representative sample of the scientific software community as possible. We sent the survey out to a number of scientific software mailing lists to which we had access, including: several internal Sandia National Labs lists (Charon, Alegra, SIERRA, Xyce, Dakota), the Trilinos users and developers lists, the PETSc users and developers lists, the Consortium for Advanced Simulation of Light Waters Reactors (CASL) members list, and the Numerical Analysis Digest mail list. While we chose these lists as a convenience sample, we believed that they would target a representative subset of the scientific software community.

The survey received 141 respondents. This sample was fairly diverse and experienced based upon four key demographics. First, in terms of their highest degree, 82% respondents held a Ph.D. with an additional 16% percent holding a Master's degree. Second, in terms of the discipline of their highest degree, the most common fields were Mathematics & Statistics, Engineering, and Computer Science. Third, Figure 3.1 shows the distribution among employer types. Finally, Figure 3.2 shows the length of time respondents had been developing scientific software.

3.2.2.2 Knowledge Source

Figure 3.3 shows that close to 60% of respondents obtained their software engineering knowledge most often from their own experience, with only a very small fraction using the more traditional approach of attending courses. While the respondents may have been able to gain

المنارة للاستشارات
www.manaraa.com

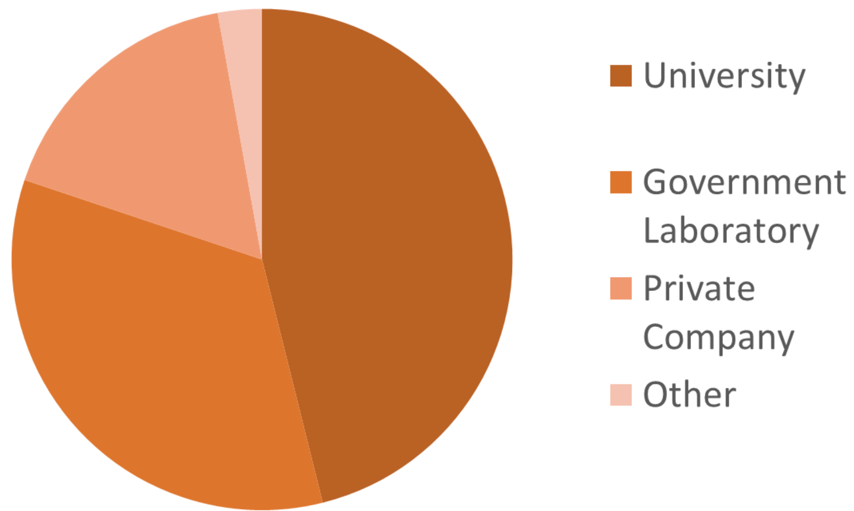


Figure 3.1: Type of Employer

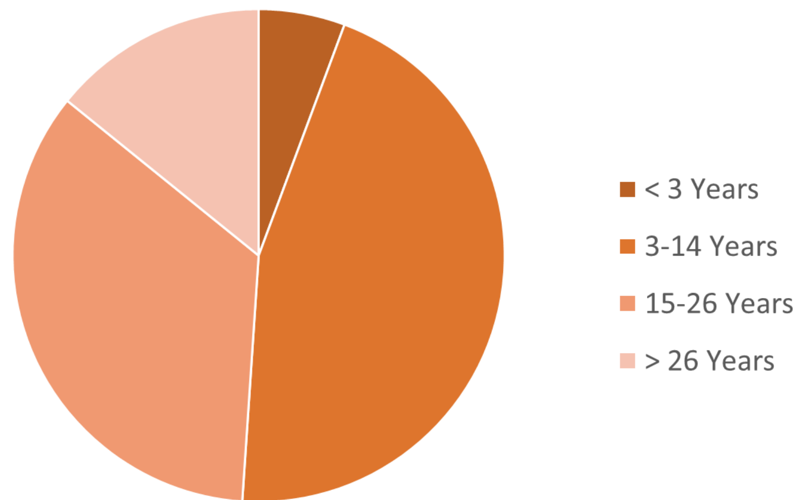


Figure 3.2: Years Developing Scientific Software

للاستشارات المنارة
www.manaraa.com

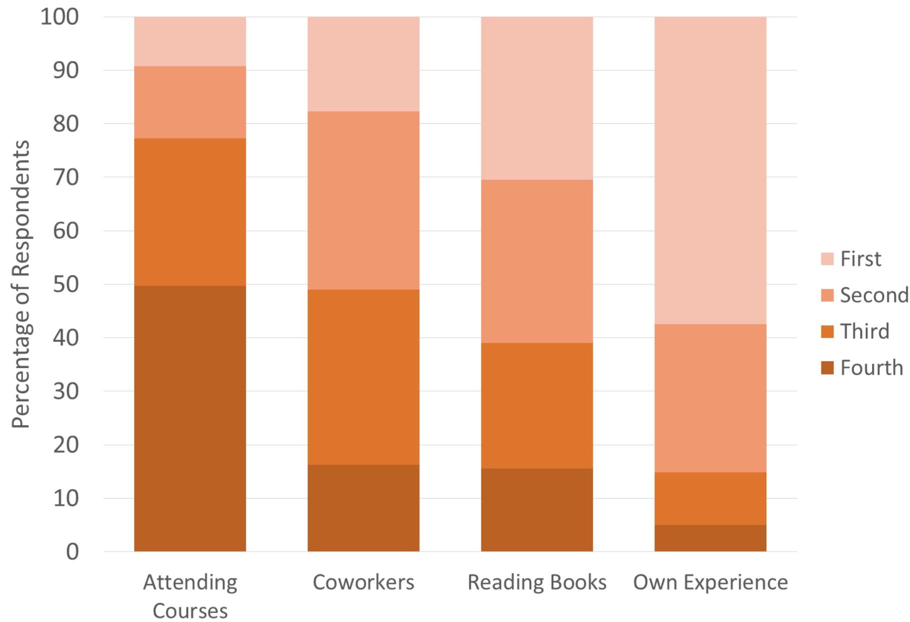


Figure 3.3: Knowledge Sources

significant software engineering knowledge in this manner, it is likely that this knowledge is limited to the subset of the relevant software engineering practices that happened to be in use within their particular environment. The lack of formal, classroom training may result in a lack of exposure to the breadth of knowledge and practices available within software engineering. Because of this lack of exposure, there are likely some useful software engineering practices with which the respondents are unfamiliar. Note that while we asked respondents to rank only one source as "first," due to the constraints of the survey tool, we were not able to enforce this requirement. Therefore, the sum for "First" in Figure 3.3 is greater than 100%.

3.2.2.3 Self-Rating of Software Engineering Knowledge

Figure 3.4 shows that the majority of the respondents indicated that their own knowledge, as well as their team's knowledge and the community's knowledge, was *mostly sufficient* for their development tasks. Interestingly, as the unit of analysis moved further away from the respondents (i.e. self to team to community) their perception about the sufficiency of software engineering

المنارة للاستشارات
www.manaraa.com

Figure 3.4: Sufficiency of Software Engineering Knowledge (First Survey)

knowledge decreased. This observation is evident in both the *no* column (increasing as the unit of analysis becomes more distant) and in the *mostly* column (decreasing as the unit of analysis becomes more distant). These results suggests that the respondents believed that they, their teams, and the community know enough about software engineering to perform their work. It is interesting to note that, in general, the respondents viewed themselves as being more knowledgeable than the rest of their team or the larger community.

In the respondents' explanations for their ratings of the sufficiency of software engineering knowledge within the scientific software community, four primary problems emerged:

- Rework,
- Performance issues,
- Regression errors, and
- Forgetting to fix bugs that were not tracked.

Notably, each of these problems has been addressed by one or more of the software engineering

المنارة للاستشارات
www.manaraa.com

practices included on the survey. Proper usage of *Requirements*, *Documentation*, *Verification and Validation*, and *Structured Refactoring* make it easier to rework existing software. Performance issues can be greatly alleviated by using proper software *Design*. *Regression* and *Integration Testing* can prevent regression errors. Finally, the issue of forgetting untracked bugs can be helped by utilizing *Issue Tracking*, *Unit Testing*, *Test-Driven Development*, and *Code Reviews*. Survey 2 (Section 3.3) explores these four problems in more depth.

3.2.2.4 Knowledge Vs. Familiarity

This section is divided into an analysis of the data on the personal level and on the team level.

Personal Level To determine whether a respondent's rating of their general software engineering knowledge correlated with their familiarity of various practices, we performed a series of Pearson's Chi-square tests. A significant result indicates that the two variables under study are dependent, i.e. they are related. In this case we used $\alpha < .05$ to judge significance. The second column of Table 3.2 shows the χ^2 and p-values for each practice. First, to get an overall sense of this relationship, the second row *All Practices*, which combines the ratings for all practices into a single variable, showed a significant relationship between knowledge and familiarity. Examining each practice in detail shows that there is a significant relationship between *personal software engineering knowledge* and *practice familiarity* for only a subset of the practices:

- Software Lifecycles,
- Requirements,
- Basic Design,
- Test-Driven Development, and

المنارة للاستشارات
www.manaraa.com

- Structured Refactoring.

To understand the potential source of non-significant relationships, we examined the distributions of the responses for those practices. The data shows that for each of these practices (except for Version Control), most respondents who rated themselves as having *mostly* or *fully* sufficient software engineering knowledge indicated *None* or *Low* familiarity with the practice. That is, if we compare the distribution of overall software engineering knowledge to the distribution of the level of familiarity, the familiarity distribution is skewed towards the Low end of the scale. This skew is more pronounced as the χ^2 value decreases and the p-value increases. In the case of Version Control, the skew was reversed with more than half of the respondents indicating *High* or *Very High* familiarity. It is interesting to note that the practice with which the respondents appeared to be most familiar, Version Control, does not specifically address any of the problems noted in Section 3.2.2.3.

This finding is particularly interesting because one of the most frequent problems reported in the scientific software development literature is difficulty with validation and verification [3, 4, 7, 10, 11, 16, 20]. Despite the importance of this problem, the respondents who considered themselves to be knowledgeable about software engineering did not report high levels of familiarity with the testing and V&V practices that would provide the most help.

Team Level The respondents' views of their team's level of knowledge (Column 3 in Table 3.2) conformed much more closely to their ratings of their team's familiarity with the individual practices with only the following practices failing to show a significant relationship:

- Integration Testing,
- Acceptance Testing, and

المنارة للاستشارات
www.manaraa.com

Table 3.2: Knowledge and Familiarity Results - shaded cells indicate significance at the $\alpha < .05$ level

Practice	Personal SE Knowledge $\rightarrow$ Practice Familiarity	Team SE Knowledge $\rightarrow$ Practice Familiarity
All Practices	$\chi^2 = 133.936; p < .001$	$\chi^2 = 212.724; p < .001$
Software Lifecycles	$\chi^2 = 18.048; p = .021$	$\chi^2 = 20.285; p = .009$
Documentation	$\chi^2 = 15.403; p = .052$	$\chi^2 = 15.549; p = .049$
Requirements	$\chi^2 = 21.294; p = .006$	$\chi^2 = 36.858; p < .000$
Basic Design	$\chi^2 = 25.302; p = .001$	$\chi^2 = 29.201; p < .000$
Intermediate Design	$\chi^2 = 13.832; p = .086$	$\chi^2 = 29.556; p < .000$
Verification and Validation	$\chi^2 = 15.469; p = .051$	$\chi^2 = 27.301; p = .001$
Unit Testing	$\chi^2 = 13.018; p = .111$	$\chi^2 = 25.565; p = .001$
Integration Testing	$\chi^2 = 10.648; p = .222$	$\chi^2 = 14.892; p = .061$
Acceptance Testing	$\chi^2 = 7.385; p = .496$	$\chi^2 = 10.797; p = .213$
Regression Testing	$\chi^2 = 11.549; p = .172$	$\chi^2 = 12.898; p = .115$
Version Control/ Change Management	$\chi^2 = 5.334; p = .721$	$\chi^2 = 21.636; p = .006$
Issue/Bug Tracking	$\chi^2 = 10.373; p = .240$	$\chi^2 = 21.647; p = .006$
Test-Driven Development	$\chi^2 = 19.511; p = .012$	$\chi^2 = 18.765; p = .016$
Structured Refactoring	$\chi^2 = 15.793; p = .045$	$\chi^2 = 19.525; p = .012$
Code Review	$\chi^2 = 18.843; p = .016$	$\chi^2 = 18.809; p = .016$
Agile Methods	$\chi^2 = 7.418; p = .492$	$\chi^2 = 20.510; p = .009$

www.manaraa.com

- Regression Testing.

In further analyzing the data, it became clear that the reason that more practices showed a significant relationship between *knowledge* and *familiarity* at the team level than at the individual level was because respondents rated team software engineering knowledge lower than individual software engineering knowledge. In other words, this result was not due to respondents indicating that their teammates were more familiar with the practices than they were individually. Despite the overall higher level of agreement, the testing practices still did not show a significant level of agreement. Further analysis of the three testing practices that were not significantly related to software engineering knowledge showed that the respondents overall saw their team as having relatively less familiarity with the testing practices. Together, these findings suggest scientific software developers could benefit from more familiarity with the appropriate software engineering practices.

3.2.2.5 Familiarity and Relevance Vs. Use

Similar to the previous analyses, for each practice, we compared the respondents *familiarity* to their level of *use* and their perception of *relevance* of that practice. The results of χ^2 tests showed a significant relationship for *relevance* and for *use* for each practice ($p < .001$). This finding suggests that scientific software developers are using the practices with which they are familiar. It also suggests that they are not being forced to use practices that they do not believe are important.

3.2.2.6 Type of Developer Vs. Familiarity/Relevance/Use

There are two types of scientific software. *Research software* is developed primarily for publishing a paper. *Production software* is developed primarily for use by others. Figure 3.5 shows the percentage of respondents who devoted various fractions of their effort to production software

المنارة للاستشارات
www.manaraa.com

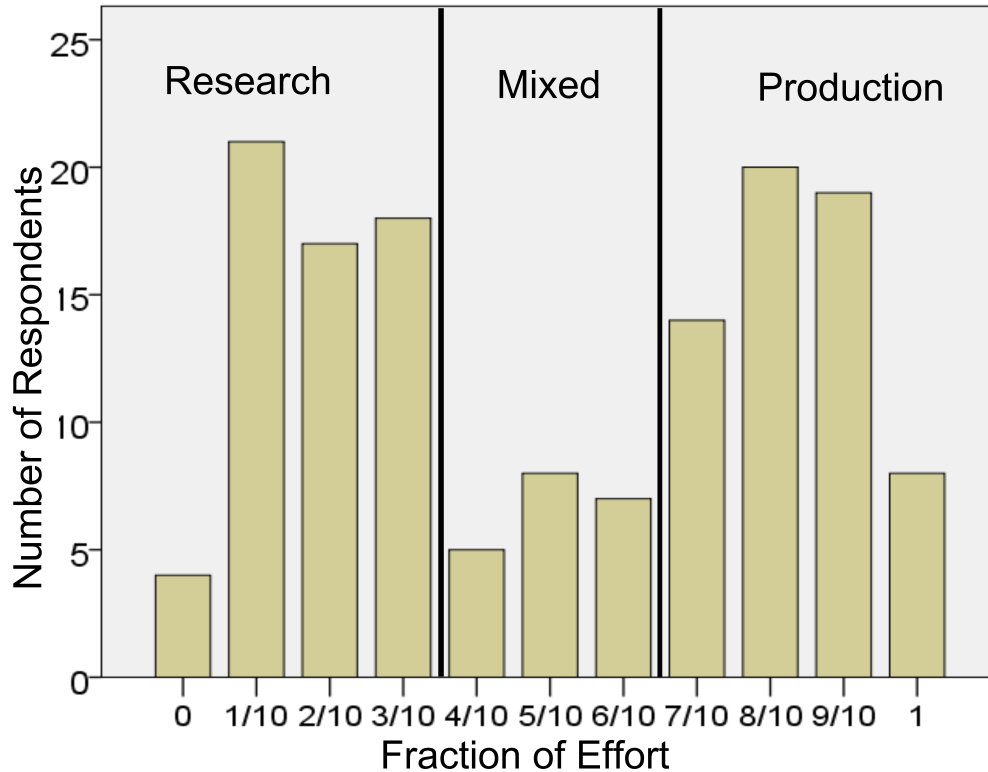


Figure 3.5: Research vs. Production (First Survey)

(i.e. 0% is purely research while 100% is purely production). Based on this data, we divided the sample into three groups (*research*, *production*, and *mixed*) for the analysis. We were surprised to see a bimodal distribution, relatively few developers in the sample split their time evenly between developing research and production software.

Our experiences led us to hypothesize that developers of research software would use a few lightweight practices while developers of production software would use practices similar to those used by traditional software developers (due to the needs of external users). Therefore, we expected that survey responses would differ based upon the type of software developed by the respondent.

Figure 3.6 shows a surprising result that the type of software a respondent develops does not seem to make a difference in the level of overall familiarity with software engineering practices.

المنارة للاستشارات
www.manaraa.com

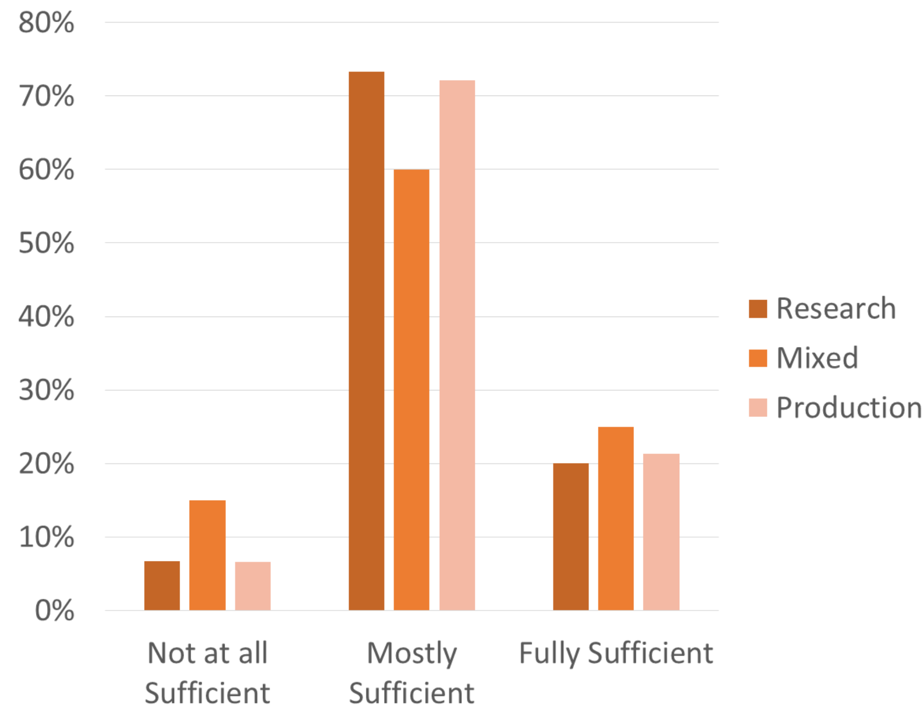


Figure 3.6: Distribution of knowledge based on development type (First Survey)

This result suggests that developers of research software may be overrating their own knowledge of software engineering.

For each practice, Figure 3.7 summarizes the responses for personal familiarity, relevance and use, broken down by developer type. Overall, production developers gave the highest ratings, followed by mixed developers, then by research developers. Despite this difference, the gap between the scores varied substantially across practices with the mixed group averaging near one extreme or the other on most practices. The practices viewed as most relevant overall were *Version Control*, *Documentation*, and *Verification & Validation*.

Interestingly, despite *documentation* requiring a significant amount of effort and potentially providing less help to the creators of the software than the users, research developers viewed it as the most relevant practice. A potential explanation for this result is a research developer might

المنارة للاستشارات
www.manaraa.com

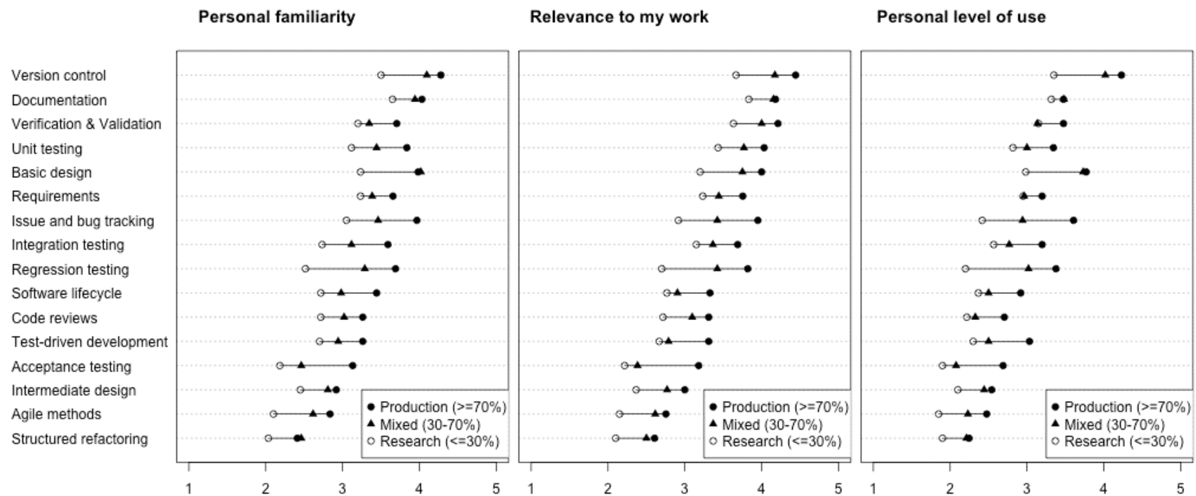


Figure 3.7: Summary of practices (First Survey)

understand documentation at a different level of detail than a production developer. For example, the amount of documentation that may be sufficient for a research-oriented developer may be light enough that they view the limited benefits as being worth the lower additional work.

For each practice, we used a χ^2 test to determine whether there was a relationship between a developer type (production, mixed, or research) and relevance. A significant relationship indicates that developer type can be used to predict the relevance of a given practice. The relationship is significant ($\alpha < .05$) for all practices except: *Documentation*, *Integration testing*, *Test-driven development*, and *Agile methods*. When combined with the results in Figure 3.7, we can conclude that production developers were statistically more likely to view most software engineering practices as relevant.

The practices that had the largest spread between the level of relevance for the research-oriented developers and the production-oriented developers were *Issue & Bug Tracking*, *Regression Testing*, *Acceptance Testing*, *Intermediate Design*, and *Version Control*. In retrospect, it is not too surprising that these practices showed such a large difference. In the case of *Issue & Bug Track-*

www.manaraa.com

ing, research-oriented software is often short-lived and used by its developers, so it is easy for developers to believe it is not as important to track bugs formally. Similarly, *Acceptance Testing* is not relevant to research-oriented software since that software is not meant for widespread external usage. *Regression Testing* and *Version Control* are also less useful for any research-oriented developer who regards software as a disposable prototype and does not intend to re-use the software.

3.2.3 Threats to Validity

This section describes the internal, construct and external validity threats for this survey.

Internal Validity: Our approach of using a convenience sample led to the potential for *Selection Bias*. Based upon the distribution of the degrees of the respondents, the mailing lists we used were not as broadly representative of the scientific community as they could have been. Further, the possibility exists that the high levels of self-rated general software engineering knowledge may have occurred because the developers who felt their software engineering knowledge was insufficient were less likely to participate. If this scenario occured, then our results represent the "best-case" scenario, with the current level of community knowledge potentially being even lower.

Construct Validity: The survey did not provide definitions of the software engineering practices. The respondents may have answered the questions based on a different definitions than the one we assumed. If this situation occurred, then the results of the survey would be less reliable. Also, the survey asked the respondents to rate their knowledge of software engineering practices in general using a three-point scale rather than a more standard five-point scale. The limited number of scale choices could have led respondents to give the middle answer more often than if they had more answer choices.

External Validity: Among the top three disciplines of respondents' degree, two were Mathematics and Computer Science. This distribution of respondents may not be representative of the

المنارة للاستشارات
www.manaraa.com

larger scientific software community. As a result, the findings here may not be appropriate for all scientific developers.

3.2.4 Conclusions

These results allow us to draw a number of conclusions that validate many of our personal observations as well as reports from a prior survey [8]. A vast majority of respondents believed that their software engineering knowledge and skills was at least "mostly sufficient" to achieve the goals of their projects, however the personal level of knowledge and use of many "best practices" from software engineering is relatively low. In addition, more than $1/3$ of the respondents thought that the skills of the overall scientific software community were not adequate to advance scientific software development.

This result appears to constitute a serious self-diagnosed problem within the scientific software community. However, while the developers were relatively successful at estimating their overall level of software engineering knowledge (i.e. row 2 of Table 3.2), they were less knowledgeable about a number of important practices, including all of the testing-related practices. We found that developers are using the practices that they are both familiar with and feel are relevant to their work. However, even the developers who see themselves as the most knowledgable in general are not using software engineering practices that would help with many of the most frequently encountered problems.

We found that the type of software a respondent develops is a useful predictor of whether they viewed a particular practice as relevant. In particular, *Issue & Bug Tracking*, *Regression Testing*, *Acceptance Testing*, *Intermediate Design*, and *Version Control* showed large differences in the perception of usefulness between the Research and Production Developers.

However, this survey did have a number of weaknesses. First, it is possible that the respon-

المنارة للاستشارات
www.manaraa.com

dents defined software engineering practices in a way different from our assumption. Second, a significant number of developers were not using practices they saw as relevant, but we had no way to gather information to understand this occurrence. Third, while we found the list of problems given in Section 3.2.2.3, we did not have enough information to rank them in terms of frequency and severity. Survey 2 addresses many of these shortcomings.

3.3 Survey 2

To gather more information about the use of software engineering practices in scientific software development and to address some of the validity threats of the first survey, the goals of the second survey were to:

- Expand and diversify the set of respondents,
- Evaluate the level of agreement with standard definitions for software engineering practices,
- Understand why developers did not use practices they found to be relevant,
- Analyze any effects of programming language on the use of software engineering practices,
- Prioritize the common problems reported by survey respondents, and
- Gather information about where scientific developers need the most support.

3.3.1 Design

To address these goals, Survey 2 has six parts. The complete version of the survey is available at: `http://carver.cs.ua.edu/Data/Journals/CSE-Surveys/survey2.pdf`.

To address the primary external validity threat from Survey 1, namely that the sample was heavily drawn from the Mathematics & Statistics and Computer Science domains, we targeted a broader population with the second survey. In addition to the mailing lists used in Survey 1, we sent the second survey to the Numerical Analysis Digest mail list, the CSGF Alumni list, the

www.manaraa.com

NERSC list, the NCCS list, the SciComp and CSMD lists at Oak Ridge National Laboratory, the NICS list, the NEAMS list, HPC-Announce, and the SIAM scientific software list. To determine whether the respondent pool was broader than in Survey 1, the second survey contained the same demographic questions. To obtain a more granular picture of the respondents' views of their own software engineering knowledge, we expanded the 3-point scale used in Survey 1 to a 5-point (1 - *not at all sufficient* to 5 - *entirely sufficient*).

Second, a threat to *construct validity* from Survey 1 was our assumption that scientists would use traditional definitions for software engineering practices. Based on further discussions with members of the scientific community, we were concerned that this assumption may have been flawed. To test this assumption, the survey asked respondents whether they agreed with a standard definition (provided on the survey) of each practice. In the case where a respondent disagreed with the given definition, he or she was given the opportunity to explain their disagreement. For the sake of consistency in data analysis, we asked the respondents to answer the remaining survey questions using the provided definition of each practice (if it differed from their own defintion).

Third, to better understand why developers did not use practices they found relevant and to account for possible validity threats in Survey 1, this survey asked respondents to rate the level of relevance and use for each software engineering practice on a 5-point scale (rather than a 3-point scale). For any practice a respondent rated relevance two or more points higher than use, the survey asked for an explanation of the discrepancy. The software engineering practices included on the second survey are the same as in Survey 1 (Table 3.1), with the following exception. To address a concern that two terms were not clearly distinguishable in Survey 1, we renamed *Basic Design* to *Low-Level Design* and *Intermediate Design* to *High-Level Design/Architecture*.

Fourth, to provide a more complete picture of the development environment and understand

المنارة للاستشارات
www.manaraa.com

why software engineering practices may have been underused, the survey investigated the effects of programming language. The common belief is that scientific software developers overwhelmingly use Fortran. Because there are fewer software engineering tools for Fortan than for other languages, like C++ or Java, developers may use fewer software engineering practices. Based on the languages commonly used by scientists, this question had the following answer choices: C, C++, Fortran, Python, Matlab, Java, Haskell, Visual Basic, C#, Perl, and Mathematica, as well as an option to write in other languages.

Fifth, Section 3.2.2.3 described four common problems faced by scientific developers: *rework*, *performance*, *regression errors*, and *untracked bugs*. To better understand the relative importance of each problem, this survey asked the respondents to rank the relative frequency and the relative severity of these problems.

Finally, to characterize the current state of software engineering practice adoption in the scientific software community, the survey had two questions. First, the survey asked respondents to describe the primary barriers faced by scientific software developers in regards to adopting software engineering practices. Second, the survey asked the respondents to identify the most important software engineering/development topics they wanted to learn.

3.3.2 Analysis

This section is organized around the goals presented in the study design. First, Section 3.3.2.1 analyzes the respondent demographics to determine whether the sample is broader than in Survey 1. Second, Section 3.3.2.2 analyzes the level of agreement the respondents had with the provided definitions of software engineering terms. Third, Section 3.3.2.3 analyzes the respondents' familiarity with, use of, and perceived relevance of the individual software engineering practices. Of particular interest is why developers do not use practices they view as being relevant.

المنارة للاستشارات
www.manaraa.com

Fourth, Section 3.3.2.4 examines the programming languages used by the respondents and whether those languages affect the use of software engineering practices. Finally, Section 3.3.2.5 analyzes the respondents' views of the frequency and severity of the common problems identified in Survey 1 (Section 3.2.2.3) along with the software engineering practices likely to address those problems.

3.3.2.1 Demographics

This survey received 151 responses. The respondent demographics indicate a larger and more diverse sample than in Survey 1. Regarding the highest degree obtained, 80.1% of the respondents held a Ph.D. while 15.2% held a Master's. The four most common academic fields in which people earned their highest degree were Engineering, Mathematics & Statistics, Physical Sciences, and Computer Science. As in Survey 1, most respondents worked at a university. Unlike Survey 1, which had a bimodal distribution between research and production developers, as shown in Figure 3.8, the respondents to Survey 2 were more heavily weighted towards research software.

3.3.2.2 Definition of Practices

Somewhat surprisingly, the survey results indicated that less than five percent of respondents disagreed with any definition, save two: *Software Lifecycles* (31 disagreements) and *Agile methods* (18 disagreements).

The definition for *Software Lifecycles* was: "A structure imposed on the development of a software system in order to ensure higher software quality." The disagreements fell into two categories. First, some respondents thought the term should be more generic, that is, a description of what happens over the life of a project rather that something that can be imposed or affects the quality of the produced software. Second, some respondents thought either that the term "quality" was too generic to encompass the goals of an explicit lifecycle model or that there were other goals for using a lifecycle model in addition to quality.

المنارة للاستشارات
www.manaraa.com

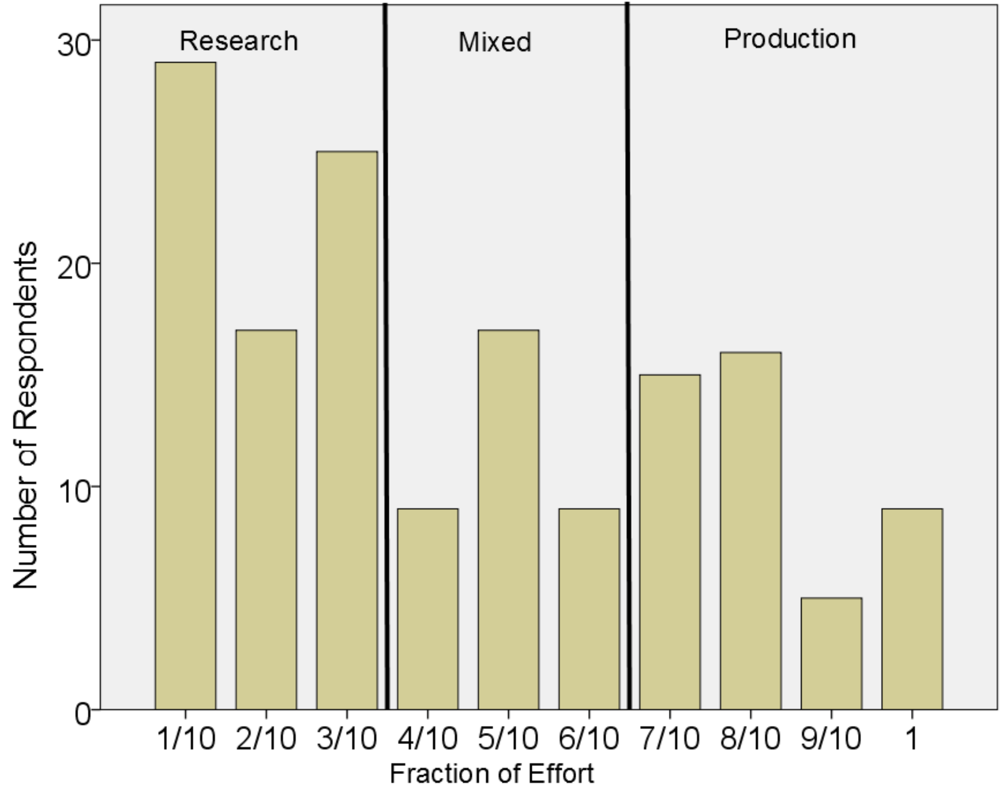


Figure 3.8: Research vs. Production (Second Survey)

The definition for *Agile Methods* was: "A group of software methodologies (e.g., Extreme Programming, Scrum, Kanban) that focus on team member interaction, always having working software throughout the lifecycle, close customer collaboration, and responding to change." The most common reason respondents gave for their disagreement was that they were simply not familiar with the term. As a side note, lack of familiarity is not really a disagreement. Only 4 disagreements were for reasons other than a lack of familiarity.

3.3.2.3 Individual Practice Analysis

In addition to analyzing the respondents' agreement with the given definitions, we also repeated the analysis from Survey 1 in terms of overall knowledge, familiarity with the practices, use of the practices, and the relevance of the practices to the respondents' work.

www.manaraa.com

Knowledge Vs. Familiarity To determine whether a respondent's rating of their general software engineering knowledge correlated to their familiarity with various practices, we performed a series of Pearson's Chi-squared tests. A significant result indicates that the two variables under study are dependent, i.e. they are related. In this case we used $\alpha < .05$ to judge significance. Table 3.3 shows the χ^2 and p-values for each practice. First, to get an overall sense of this relationship, the second row *All Practices*, which combines the ratings for all practices into a single variable, showed a significant relationship between knowledge and familiarity. Examining each practice in detail shows that there is a significant relationship between **personal software engineering knowledge** and **practice familiarity** for only *Software Lifecycles* and *Code Reviews*. To understand the lack of significant relationships in more detail, we examined the distributions of the responses. The data shows that for each of these practices, most respondents rated themselves as having Medium or High familiarity with the practice, regardless of how they rated their overall software engineering knowledge. That is, if we compare the distribution of overall software engineering knowledge to the distribution of the level of familiarity, the familiarity distribution is skewed towards the High end of the scale. This skew is more pronounced as the χ^2 value decreases and the p-value increases.

Familiarity/Relevance Vs. Use We expected that each respondent's view of their own familiarity with a practice and the relevance of that practice would once again be good predictors for use of the practice. The Chi-squared analysis confirms the relationship between familiarity and relevance found in Survey 1 (Section 3.2.2.5). Once again, we found that scientific software developers are using the practices that they are familiar with and not being forced to use practices that they do not see as relevant. Figure 3.9 summarizes the 382 responses to the question of why respondents did not use a practice that they viewed as relevant. Notably, the largest group of responses falls

المنارة للاستشارات
www.manaraa.com

Table 3.3: Knowledge and Familiarity Results - shaded cells indicate significance at the $\alpha < .05$ level

Practice	Personal SE Knowledge $\rightarrow$ Practice Familiarity
All Practices	$\chi^2 = 76.184; p < .000$
Requirements	$\chi^2 = 14.577; p = .265$
Documentation	$\chi^2 = 12.363; p = .194$
Verification & Validation	$\chi^2 = 7.361; p = .600$
Structured Refactoring	$\chi^2 = 10.447; p = .577$
High Level Design	$\chi^2 = 19.936; p = .068$
Low Level Design	$\chi^2 = 14.982; p = .242$
Regression Testing	$\chi^2 = 13.580; p = .328$
Integration Testing	$\chi^2 = 19.563; p = .076$
Issue Tracking	$\chi^2 = 10.863; p = .541$
Unit Testing	$\chi^2 = 17.519; p = .131$
Test-Driven Development	$\chi^2 = 14.769; p = .254$
Code Reviews	$\chi^2 = 23.082; p = .027$
Software Lifecycles	$\chi^2 = 38.704; p < .001$

under "Little Relevance" which means that, while they rated the relevance of the practice higher than they did their usage of the practice, they still stated that the relevance was low enough to justify not using the practice.

Type of Developer Vs. Familiarity/Relevance/Use As in Survey 1, we expected to find a binary distribution between research and production developers. Instead, as shown in Figure 3.8, the respondents to the second survey considered themselves primarily research developers. Based on Figure 3.10, it appears that the production developers in Survey 2 were slightly more likely to view their level of software engineering knowledge as *Mostly Sufficient* or *Fully Sufficient* than the research developers, however this difference is not significant. This lack of relationship suggests that our earlier belief, stated in Section 3.2.2.6, that production developers would be more likely to view software engineering practices as relevant may not be true.

For each practice, Figure 3.11 summarizes the responses for personal familiarity, relevance,

المنارة للاستشارات
www.manaraa.com

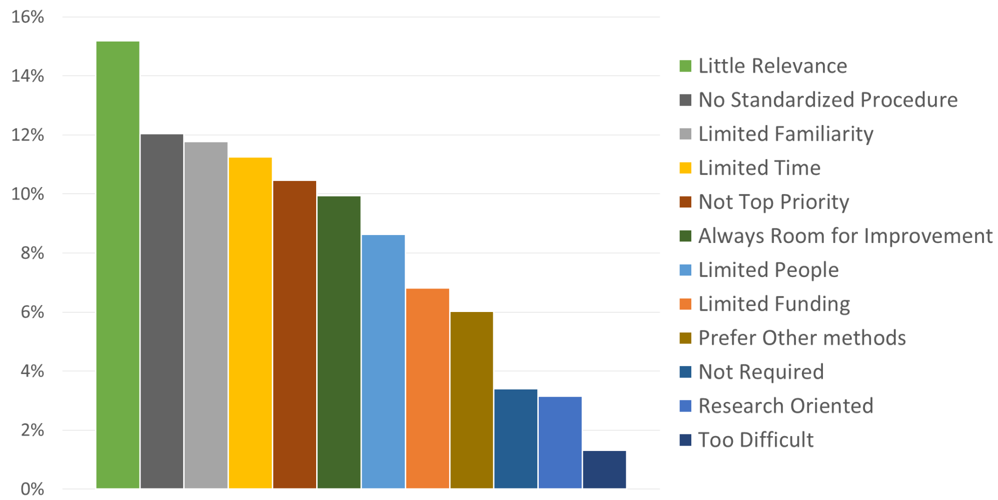


Figure 3.9: Reasons why developers did not use practices they felt were relevant

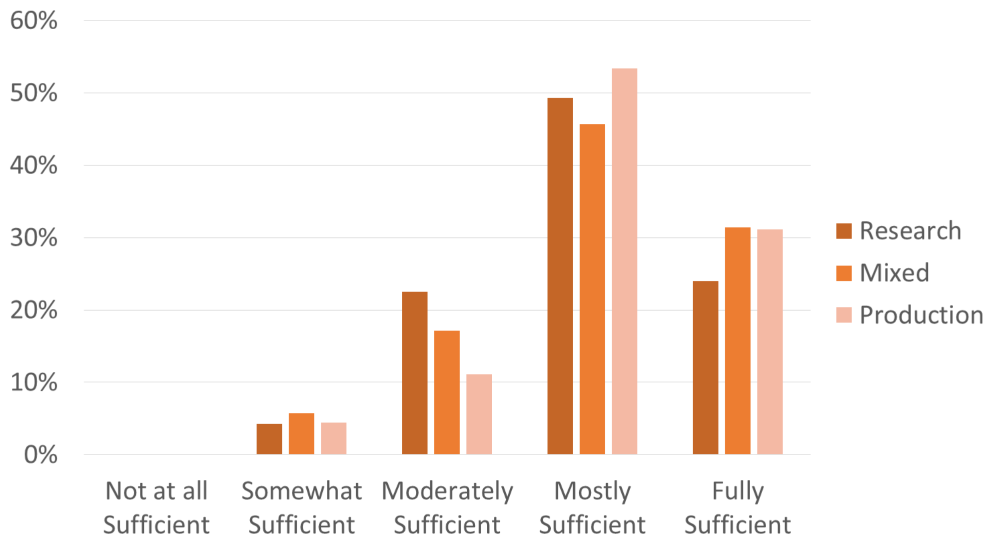


Figure 3.10: Distribution of knowledge based on development type (Second Survey)

المنارة للاستشارات
www.manaraa.com

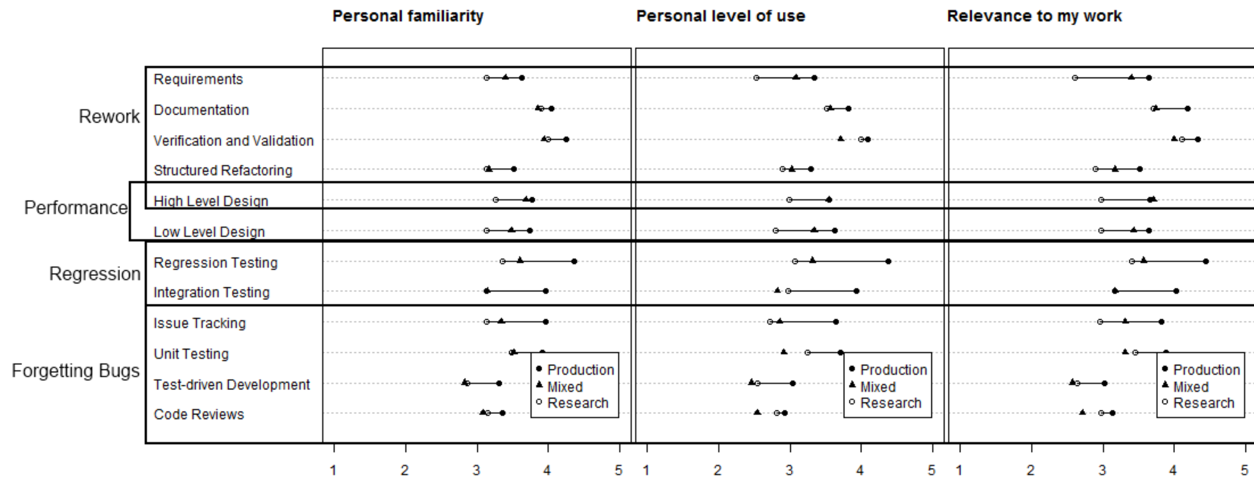


Figure 3.11: Summary of practices (Second Survey)

and use, broken down by developer type. As before, in every case production developers were the most familiar with and most likely to use the software engineering practices. However, unlike in Survey 1, the mixed developers came in slightly lower than the research developers in terms of knowledge and use in a few cases.

3.3.2.4 Languages

While we expected Fortran would be the most commonly used language, as Figure 3.12 shows, C++ is slightly more popular than Fortran, with C, MATLAB, and Python trailing close behind. In addition, most respondents used multiple languages (an average of three). These results suggest one of two things. Either scientific software development may be shifting away from its Fortran dominance or the survey respondents were not representative of the general scientific software community. If the first case is true, it would be promising, as software engineers have developed many more tools for C++ and Python than for Fortran.

Regarding whether language affected the use of software engineering practices, we hypothesized that developers who use Fortran would be less likely to use the various software engineering

المنارة للاستشارات
www.manaraa.com

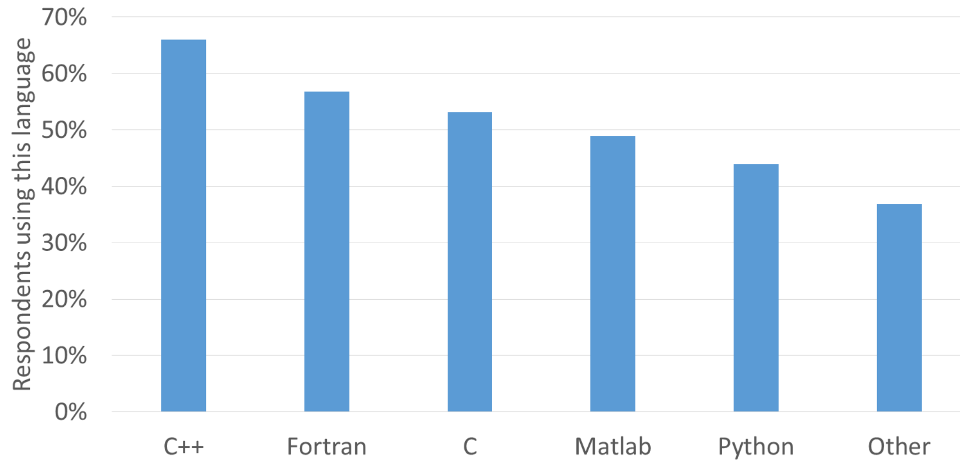


Figure 3.12: Languages used in scientific software

practices than developers who did not use Fortran. To test this hypothesis, we divided the sample into two groups: (1) the respondents who used Fortran and (2) the respondents who did not use Fortran. In order to determine if there was any effect on practice usage based upon programming language, we performed a Pearson's Chi-squared test to compare the distribution of responses about each practice between these two groups. The results showed that there was no significant difference between the groups for any of the practices. Therefore, use of Fortran is not a significant predictor of whether a developer will use various practices. Note that the survey did not ask respondents to link specific practices to programming languages. Therefore, for the respondents who used multiple languages, we cannot determine which practices went with which languages.

3.3.2.5 Primary Problems

In terms of the most frequent types of problems, the respondents ranked *rework* first, followed by *performance issues*, *regression errors*, and finally *forgetting to fix untracked bugs*. To better evaluate whether the respondents really understood how various software engineering practices could help address these problems, we mapped each software engineering practice to one of the four problems. We performed this mapping based upon our own experience with traditional

المنارة للاستشارات
www.manaraa.com

software engineering environments. Figure 3.11, discussed throughout this section, maps the software engineering practices to the reported problems. The remainder of this section provides a justification for the mapping.

Rework Rework was the most frequent problem reported by the respondents. Research and production developers both tended to view **rework** as a moderately to highly frequent problem. Both groups of developers also tended to view the severity of **rework** to be moderate to high. Five software engineering practices included on the survey should either reduce the need for rework or reduce the effort required to perform rework: *requirements*, *verification & validation*, *high-level design*, *documentation*, and *structured refactoring*.

Proper usage of *requirements* helps reduce the frequency of **rework** by determining what the software needs to do prior to implementation, thereby reducing the chances of unanticipated changes. Additionally, in an agile environment, like most scientific software projects, the proper use of requirements helps developers document and manage evolving requirements. The process of creating and using requirements is highly relevant and moderately used by production developers. Conversely, requirements have low relevance and low usage by research developers. The most common reasons that respondents gave for not creating *requirements* were:

- They had limited people to create the requirements,
- They felt that there was no need to create specific requirements for their software, and
- They have no standardized procedure for producing requirements.

The use of *verification & validation* practices throughout the development process decrease the likelihood of **rework**. By identifying and removing problems during development, fewer problems must be fixed via rework. *Verification & validation* practices were among the highest ranked

المنارة للاستشارات
www.manaraa.com

practices overall, with both research developers and production developers reporing high or very high relevance and use. The main reasons that respondents gave for not performing *verification & validation* were:

- They had too little time to verify and validate their software, and
- They felt that verification & validation had little relevance to their work.

Use of *high-level design* can help prevent the need for **rework**. Similar to the use of requirements, proper design will help developers understand the system and therefore should reduce the chance that developers will introduce defects requiring rework. Once again, high-level design showed a disparity between research developers and production developers in both use and relevance. Research developers viewed high-level design as having a moderate level of relevance, while production developers viewed it as having a high level of relevance.

Unlike the previous three practices, proper *documentation* does not necessarily prevent the need for **rework**. However, the presence of complete and accurate documentation, which provides readily-available information about the software, can significantly lower the amount of effort required to perform rework. Documentation is also among the most relevant and used practices in for both the research and production developers. The most common reasons that respondents gave for not creating *documentation* were:

- They had too few people to create the documents,
- They had too little time to create the documents,
- They felt that documentation was not needed because they were primarily focused on research, and
- They had no standardized procedure for producing requirements documentation.

المنارة للاستشارات
www.manaraa.com

Finally, *structured refactoring* both prevents the need for in-depth **rework** and reduces the effort required to perform **rework**. It prevents the need for rework by allowing developers to fix design deficiencies discovered after the design is finalized without changing the functionality of the software. Additionally, if refactoring is used to reduce software complexity and increase maintainability, then it can also reduce rework effort. In this case, the production developers reported only a slightly higher level of usage than did research developers. However, production developers considered structured refactoring to be noticeably more relevant than did research developers. The primary reasons that the respondents gave for not performing *structured refactoring* were:

- They did not view structured refactoring as a top priority and
- They had no standardized procedure for performing structured refactoring.

The benefits of these practices are not readily obvious. The first step forward in addressing this problem is to better inform scientific software developers about how utilizing these practices can reduce the need for expensive rework. Secondly, one of the main reasons given for not using half of the practices was that they had no standardized procedure to perform the practices. To address this problem, software engineers will need to work with scientific software developers to produce these standardized procedures.

Performance Issues Survey respondents reported performance issues as the second most common problem. However, they considered performance issues to be somewhat less severe than the other problems. Research and production developers viewed performance issues to be of a moderate frequency, but production developers viewed it as being significantly more severe ($t = -2.846$, $p=.005$). The two software engineering practices that most directly relate to improving

المنارة للاستشارات
www.manaraa.com

performance are *high-level design* and *low-level design*. In both cases design decisions can either optimize performance or slow down performance.

High-level design, or architecture, affects global **performance** by determining how the system is organized and what types of communication must occur. Note that high-level design is also relevant for **rework**. As discussed previously, production developers viewed high-level design as being more relevant than did research developers. Many respondents who reported a low level of use of *high-level design* claimed that they had limited familiarity with it. Of those who believed they were familiar with *high-level design*, a large number saw it as having little relevance. Most of those who saw *high-level design* as having little relevance based this belief upon the fact that they were working either on small projects or with legacy code. The respondents stated that both situations force the developer to use a given architecture without being able to modify it.

Low-level design focuses more on optimizing the performance within a module through use of appropriate data structures and algorithms. Production developers reported that *low-level design* was both highly relevant to their work and highly used. Conversely, research developers reported only a medium relvance and a medium/low level of use in their projects. Many respondents also found the lack of a strong *high-level design* made it impossible for them to produce a detailed *low-level design*. The amount of upfront effort required to produce *low-level design* hampered the ability of developers to utilize the practice. These observations indicate that there is a need for scientific software developers to better understand the benefits of *low-level design*.

The low usage of *low-level design* among research developers hampers their ability to take advantage of the performance increase that could be gained from optimizing algorithms within each software module. Similar to the other practices, we must identify which *high-level* and *low-*

المنارة للاستشارات
www.manaraa.com

level design approaches are most appropriate for the scientific software domain and educate developers appropriately.

Regression Errors Survey respondents reported regression errors as the third most frequent problem. While, production developers indicated that regression errors were more problematic than research developers did (t=-2.937, p=.004), there was little difference in the frequency of regression errors. Two types of testing can help to address regression errors: *regression testing* and *integration testing*.

The goal of *regression testing* is to detect **regression errors**. Therefore, this practice is clearly the most relevant to the problem. The usage of regression testing by research developers has a wide variance around the mean shown in Figure 3.11. Conversely, production developers report a very high level of use. Research developers further tend to view *regression testing* as having a high level of relevance with most production developers rating it as very highly relevant. *Regression testing* was seen as important by almost all of the respondents who commented on their usage of it. The reasons given for not utilizing *regression testing* were largely related to time constraints or a lack of familiarity with how to perform it. Another common issue, however, was that the developers did not have an automated *regression testing* tool for a large scientific problem.

While it is not specifically intended to detect regression errors, *integration testing* can be used to verify that no new regression errors have entered the software when new code is added. Much like regression testing, integration testing was used only by some research developers. In addition, there was no real pattern in the research developers' views of the relevance of integration testing. Production developers, however, reported a high level of both usage and relevance. The respondents also saw a considerable need for *integration testing* that was hindered by a few underlying problems. In many cases, developers did not have the support for or familiarity with *unit*

المنارة للاستشارات
www.manaraa.com

testing, which is generally a prerequisite to building integration tests. Otherwise, most respondents did not do the integration of software units themselves and they assumed that the developer who did the integration performed any needed tests. These problems would also make it difficult to implement *regression testing* because it relies upon an existing test suite for maximum efficiency. *Regression testing* also requires all developers involved with the software to use it.

Addressing these issues will require two steps. First, scientific software developers need to be better trained in *unit testing*. In order to do this training, scientific software developers will need to work with software engineers to determine what is the correct size of a code "unit" to provide useful testing information. Secondly, once they are able to perform unit tests, scientific software developers will need automated regression testing tools developed to work specifically in the scientific software environment.

Untracked Bugs While this problem is the least frequent, it is the most severe. Unlike the other problems, there is no relationship between either severity or frequency and type of developer (research or production). Four software engineering practices from the survey address this problem: *issue/bug tracking*, *unit testing*, *code reviews*, and *test-driven development*.

Using *issue/bug tracking* software allows scientific software developers to track bugs with a minimal amount of extra work. Despite developer types not affecting the views of frequency and severity of this problem, there was a strong divide on the use of issue/bug tracking software between research developers and production developers. Research developers reported a low level of relevance and use, while production developers reported a high level of relevance and use. The developers who had a discrepancy between relevance and use reported four major reasons for the discrepancy:

المنارة للاستشارات
www.manaraa.com

1. They felt that it was not relevant to their development or of little use;
2. They had no standardized procedure for utilizing issue/bug tracking;
3. They prefered other methods for keeping track of bugs; and
4. Tracking issues and bugs was not their top priority.

In addition to issue/bug tracking, testing methods need to be used to detect these bugs during the development process instead of relying on the bugs being found afterwards. It is considerably more expensive to correct a bug after the software is deemed complete and in use. *Unit testing* provides the ability to check individual parts of the software without testing it in its entirety. *Unit testing* showed a high level of both relevance and use across both types of developers, with production developers viewing it as slightly more relevant and being slightly more likely to use it. The respondents highlighted three main discrepencies between relevance and use with *unit testing*:

1. Projects build on legacy code prevents them from utilizing *unit testing* to its fullest extent;
2. Use of *unit testing* creates a large amount of extra up-front work; and
3. Not familiar enough with the practice to implement it.

Code reviews are important because they provide a chance to find and fix mistakes that were overlooked in the design phase. Similar to unit testing, there was no significant relationship between the usage and relevance of code reviews and type of developer (research or production). The distribution of answers for usage and relevance for each type of developers was approximately normal. Use of *code reviews* was primarily limited by two factors. First, the developer was often not working in a large enough group to perform formal code reviews. The problem of group size is complicated by the fact that scientific developers are frequently exploring questions that are too complicated for external developers to understand the code. Second, many of the benefits of *code*

المنارة للاستشارات
www.manaraa.com

reviews are not directly obvious (i.e. determining factors that cause bugs to enter code reduces the chances of similar bugs entering later codes). This lack of recognition of the benefits results in a development team giving the review process a lower priority than other tasks. This observation is important because the additional benefits from *code reviews* result in the developers producing better code that requires less effort to test and review.

Test-driven development is a development process that is particularly well-suited to the development of complex software. It forces the developer to add one piece of functionality at a time, simplifying the testing process. Because test-driven development requires the developers to write a test even before they develop the code that will satisfy the requirements of that test, it makes it less likely for untracked bugs to enter into the software in the first place. The results for the usage and relevance of test-driven development, however, were considerably different from the other results. While research developers were as likely as production developers to use test-driven development, they viewed it as more relevant than did production developers. A number of the respondents who claimed they did not use *test-driven development* unintentionally used an approach that could be adapted to *test-driven development*. They used an iterative method of adding a feature and then testing to be sure it worked properly. It would be easy to adapt this method to *test-driven development* by simply changing the order and writing the test of the desired functionality before implementing the procedure. There were four major barriers to the adoption of *test-driven development*: not enough resources, lack of coherent high and low level designs, complicates the creation of test cases, and not familiar enough with the practice to implement it.

3.3.3 Conclusion

Based on this analysis, we can draw a number of conclusions from Survey 2. First, the respondents represented a broader sample of the scientific development community. Second, the

المنارة للاستشارات
www.manaraa.com

responses confirmed that scientific software developers generally used the same definitions for software engineering practices as do software engineers. Less than five percent of the respondents disagreed with the definition for each practice except for *Software Lifecycles* and *Agile Methods*. This finding means that a threat to validity in Survey 1 may have had minimal, if any, effect on the conclusions drawn. Third, we analyzed why developers did not use practices they viewed as being relevant and found the primary reasons were:

1. They view the practice as having little relevance,
2. They have no formal method for performing the practices, and
3. They have too limited familiarity with the practice to utilize it.

Notably, the most popular reason was that they felt the practice had little relevance despite rating the relevance of the practice higher than they did their use of the practice. This inconsistency might be because they viewed the relevance to be below a minimum threshold. Addressing these problems will require software engineers to work with scientific software developers to provide evidence of the usefulness of the practices in their environments, produce clear methods for utilizing the practices, and provide appropriate training.

Another promising finding is that many respondents already use high-level languages that have significant support from the traditional software engineering world. This result means that many tools currently exist that may, with minor tweaking, be able to help scientific software developers employ the software engineering practices most likely to help address their problems.

The remainder of this section examines the four major problems identified in Section 3.2.2.3 and discussed in Section 3.3.2.5.

المنارة للاستشارات
www.manaraa.com

Rework While rework was the most frequent problem encountered by both **research** and **production** developers, most of the practices that should help address this problem were only used at a low to medium level by **research** developers with the exception of *verification & validation*. Even more interestingly, despite recognizing that rework was a frequent and somewhat severe problem, most developers only saw these practices as being of moderate relevance to their work.

Performance Issues Performance issues tied with rework for being the most frequent problem encountered by **production** developers, but only the second most frequent for **research** developers. However, both types of developers viewed performance issues as being the least severe. Perhaps because of this relatively low severity rating, the relevance and use of the practices that should help address this problem was only at a moderate level among **research** developers. Conversely, **production** developers viewed the practices at a high level of use and relevance.

Regression Errors Both **production** and **research** developers viewed regression errors as the second most frequently encountered problem. **Production** developers viewed regression errors as the most severe problem, but **research** developers viewed it as the third most severe. As we would expect from the varying severity, **production** developers saw the relevance of the practices that address regression errors to be very high, with use also falling between high and very high. **Research** developers, on the other hand, only saw the use and relevance to be at only a moderate level.

Untracked Bugs **Research** and **production** developers viewed this problem as the first and second most severe, respectively. Despite the importance of this problem, none of the respondents who commented used a formal *issue/bug tracking* system. Instead they either used mailing lists or tried to keep track of bugs mentally. However, many of the respondents did believe that the adoption of such a system would be beneficial. Notably, both **production** and **research** developers

www.manaraa.com

viewed Test-driven Development and Code Reviews to be at a low to medium level of use and relevance.

3.3.4 Threats to Validity

While this survey eliminated the threats to external and construct validity from Survey 1, potential threats to internal validity remained as well as a new threat to construct validity.

Internal Validity: Our approach led to the potential for *Selection Bias*. While broader than the mailing lists used for Survey 1, the mailing lists we used were not neccessarily fully representative of the scientific community as they could have been. The nature of the survey as well as our desire to preserve anonymity lead us to not ask survey respondents to indicate from which mailing list(s) they received the survey invitation or if they had participated in Survey 1. If there was a large overlap between the respondents of the two surveys, then the ability to generalize results would be limited. However, the demographics of the respondents to Survey 2 suggest that this threat is not serious. Because of the need for anonymity, an important caveat is that the data reported from the survey is drawn from the opinions of the respondents and may or may not be consistent with the true state of software engineering practice in the scientific software community. Further, the possibility still exists that the high levels of self-rated general software engineering knowledge may have occurred because the developers who felt their software engineering knowledge was insufficient were less likely to participate. If this scenario occured, then our results represent the "best-case" scenario, with the current level of community knowledge potentially being even lower.

Construct Validity As noted in Section 3.3.2.4, many respondents utilized more than one language. Therefore, a threat to construct validity is that we cannot map specific practices to individual languages. It is possible, for example, that someone who indicated using both C++ and Fortran and

المنارة للاستشارات
www.manaraa.com

indicated high usage of Version Control could have used Version Control only when programming in C++. In our analysis we had to assume the use of the practice applied to all indicated languages.

3.4 Analysis and Conclusions Across Both Surveys

This section treats Survey 1 and Survey 2 as a series that, when taken together, can provide more insight than either survey individually. By replicating and expanding upon Survey 1, the results of Survey 2 are able to both confirm and extend the findings of Survey 1. In general, the results of Survey 2 confirmed the results of Survey 1. The remainder of this section provides details on the observations across both surveys.

3.4.1 Demographics

The demographics of the respondent pools differed across the surveys. Based on two key demographics, we can conclude that the respondents to the two surveys represented different segments of the scientific software population. First, while the respondents to Survey 1 were heavily weighted towards Mathematics & Statistics and Computer Science domains, the respondents to Survey 2 represented a more diverse sample (including also respondents from Engineering and Physical Sciences). Second, the distribution of the respondents between production software and research software differed between the two surveys. These observations are important because the results of the two surveys were otherwise similar. Therefore, the more diverse sample indicates that the results can generalize to the larger scientific software community with more confidence.

3.4.2 Knowledge Source

One of the important questions for any community to understand is where its members obtain knowledge about key concepts. In the more traditional software engineering world, the majority of the community members have some type of formal training in computer science or software engineering. Our anecdotal assessment of the scientific software community, prior to the

المنارة للاستشارات
www.manaraa.com

surveys, was that its members obtained their knowledge through vastly different means. The results from both surveys, which were consistent, confirmed this anecdotal conclusion. Respondents to both surveys indicated the most frequent method of gaining software engineering knowledge was to obtain it on their own. Furthermore, the respondents indicated that attending training courses (which would likely also include university courses) was the least frequent method of obtaining knowledge. While a developer can obtain some level of useful knowledge on his or her own, generally speaking that knowledge is not sufficient to meet the demands of developing complex scientific software (as indicated by the relatively low familiarity and use of a number of key software engineering practices).

3.4.3 Individual Practice Analysis

One of the most important purposes of these surveys was to get a solid picture of the current state of software engineering knowledge in the scientific software development community. One of the common issues is that developers tend to "not know what they don't know." In this case, existing software engineering practices could help scientific software developers be more effective and efficient. Our hypothesis was that scientific developers often do not know enough about software engineering to be aware of relevant practices. The results from the two surveys tended to confirm this hypothesis. In many cases, especially in Survey 1, even the respondents who thought they were knowledgeable about software engineering had little familiarity with individual software engineering practices. While the respondents to Survey 2 evidence a similar discrepancy between overall knowledge and knowledge of specific practices, they actually tended to be more familiar with the practices. However, in both surveys scientific software developers reported low rates of knowledge and use of some common testing practices. Without using effective testing procedures, software developers in any domain cannot have confidence in the accuracy of the

المنارة للاستشارات
www.manaraa.com

results produced by the software. This problem is particularly important in the scientific domain, where it is necessary for developers to determine whether the source of incorrect results is the underlying scientific theory or simply a software mistake. This type of misunderstanding could potentially allow valuable scientific results to be dismissed because of software errors rather than because of scientific problems.

3.5 Conclusion

Overall, the results from the two surveys were largely consistent. This consistency leads us to believe that, overall, software engineering practices are underused in the development of scientific software. Furthermore, scientific developers often lack the familiarity with these practices that would even allow them to make an informed decision about their relevance to the current project. Conversely, it does appear from the results that those developers whose software is typically used by external users (i.e. production software) were more likely to be using some of the important software engineering practices. We argue that one of the important causes for the lack of knowledge and use of appropriate software engineering practices is the general lack of formal training received by scientific developers. This lack of formal education limits the developers' ability to take full advantage of practices that could greatly aid their software development. One potential solution to this lack of formal education is for members of the software engineering community to make concerted efforts for outreach into the scientific software community. Another potential solution is to better document and share instances of successful use of a software engineering practice in the development of scientific software.

المنارة للاستشارات
www.manaraa.com

REFERENCES

[1] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz, "Understanding the high-performance-computing community: A software engineer's perspective," *IEEE Software*, vol. 25, no. 4, pp. 29–36, Jul. 2008.

[2] J. Carver, D. Heaton, L. Hochstein, and R. Bartlett, "Self-perceptions about software engineering: A survey of scientists and engineers," *Computing in Science Engineering*, vol. 15, no. 1, pp. 7–11, Jan. 2013.

[3] J. C. Carver, "Report from the second international workshop on software engineering for computational science and engineering," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 14–19, 2009.

[4] J. C. Carver, L. M. Hochstein, R. P. Kendall, T. Nakamura, M. V. Zelkowitz, V. R. Basili, and D. E. Post, "Observations about software development for high end computing," *CTWatch Quarterly*, vol. 2, no. 4A, pp. 33–37, 2006.

[5] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 550–559.

[6] J. C. Carver, R. Bartlett, D. Heaton, and L. Hochstein, "Self-perceptions about software engineering: A survey of scientists and engineers," Department of Computer Science, The University of Alabama, Tech. Rep. SERG-2011-04, Nov. 2011.

[7] S. M. Easterbrook and T. C. Johns, "Engineering the software for understanding climate change," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 65–74, 2009.

[8] J. Hannay, H. Langtangen, C. MacLeod, D. Pfahl, J. Singer, and G. Wilson, "How do scientists develop and use scientific software?" in *Proccedings of the ICSE Workshop on Software Engineering for Computational Science and Engineering*, May 2009, pp. 1–8.

[9] D. Heaton, J. Carver, R. Bartlett, K. Oakes, and L. Hochstein, "The relationship between development problems and use of software engineering practices in computational science & engineering: A survey," in *Proccedings of the 1st International workshop on Maintainable Software Practices in e-Science*, May 2012.

المنارة للاستشارات
www.manaraa.com

[10] L. Hochstein and V. R. Basili, "The ASC-Alliance projects: A case study of large-scale parallel scientific code development," *Computer*, vol. 41, no. 3, pp. 50–58, 2008.

[11] D. Kelly, D. Hook, and R. Sanders, "Five recommended practices for computational scientists who write software," *Computing in Science and Engineering*, vol. 11, no. 5, pp. 48–53, 2009.

[12] Z. Merali, "Computational science: Error, why scientific computing does not compute," *Nature*, vol. 467, no. 7317, pp. 775–777, 2010.

[13] L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana, "A survey of scientific software development," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 12:1–12:10.

[14] D. E. Post and R. P. Kendall, "Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: Lessons learned from asci," *International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 399–416, 2004.

[15] D. E. Post and L. G. Votta, "Computational science demands a new paradigm," *Physics Today*, vol. 58, no. 1, pp. 35–41, 2005.

[16] D. E. Post, R. P. Kendall, and R. F. Lucas, "The opportunities, challenges, and risks of high performance computing in computational science and engineering," in *Quality Software Development*, ser. Advances in Computers, M. V. Zelkowitz, Ed. Elsevier, 2006, vol. 66, pp. 239 – 301.

[17] R. Sanders and D. Kelly, "Dealing with risk in scientific software development," *IEEE Software*, vol. 25, no. 4, pp. 21–28, Jul. 2008.

[18] J. Segal, "Some problems of professional end user developers," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, 2007, pp. 111–118.

[19] J. Segal and C. Morris, "Developing scientific software," *IEEE Software*, vol. 25, no. 4, pp. 18–20, Jul. 2008.

[20] F. Shull, J. Carver, L. Hochstein, and V. Basili, "Empiricalrical study design in the area of high-performance computing (HPC)," in *Proceedings of the 2005 International Symposium on Emperical Software Engineering*, 2005, pp. 17–18.

[21] G. Wilson, "Those who will not learn from history..." *Computing in Science and Engineering*, vol. 10, no. 3, pp. 5–6, May 2008.

المنارة للاستشارات
www.manaraa.com

Chapter 4

CASE STUDIES

4.1 Introduction

Scientists and engineers use software models to replace dangerous or expensive experimentation or to conduct studies that would not be possible otherwise. These three examples illustrate how software is used in various scientific domains. First, in climate science, software models allow meteorologists to forecast the weather conditions and make predictions about dangerous weather events. Without these models, meteorologists are limited to manually examining historical weather patterns to extrapolate predictions about future weather. This historical approach is time-intensive, which is problematic in the face of the rapid pace of changing weather conditions. Additionally, the historical approach primarily gives general predictions, which means it is likely less accurate than the results given from the models. Second, in fields like earth science, a different problem emerges. Many of the phenomena occur so slowly that it is inefficient to experiment with them physically. Software models allow earth scientists to speed up the effects of their experiments. Third, in fields like nuclear science, yet a different problem emerges, the problem of safety. It is much safer for scientists to simulate the effects of nuclear reactions than to conduct a physical experiment.

As these examples highlight, scientists and engineers are increasingly reliant on the results of software simulations to inform their decision-making process. Because of this reliance, it is vital for the software to return accurate results. While the correctness of the science behind the software

المنارة للاستشارات
www.manaraa.com

is the most important factor in the accuracy of results, the correctness and quality of the software is also of primary importance. The field of software engineering provides tools and methods that help developers increase and verify software quality.

The specific areas of scientific software quality we seek to improve are maintainability and reusability. As these two qualities greatly affect the cost of developing software in traditional software engineering domains [8, 9], we hypothesize that the same would prove true in the scientific software domain. In order to evaluate this hypothesis, we will look at the three sub-areas of Readability, Preservation of Knowledge, and Testing. Improvement in each of these areas has been shown to improve both maintainability and reusability of traditional software engineering software [3, 10, 20, 44]. In traditional software engineering, code reviews have been found to have a significant impact on the readability of software and the preservation of knowledge among members of the development team [3]. Also, regression testing has been shown to be an effective method of detecting failures that have been introduced by new development on software that has previously been determined to be correct [22, 30, 31, 35, 46]. Finally, integration testing has been shown to be effective at helping developers identify errors that occur when there are miscommunications between components of a system that have been tested individually [25, 27]. In order to show that these techniques will have similar effects on scientific software development, we have formed a number of hypotheses:

- The use of peer code reviews will significantly improve the readability of scientific software.
- The use of peer code reviews will significantly increase the preservation of knowledge across a scientific software development team.

المنارة للاستشارات
www.manaraa.com

- The use of regression and integration testing in concert will provide a means for scientific software developers to show that their software is "more correct."

This paper presents the results of two examples that show software engineering techniques can be applied or modified to increase the maintainability and reusability of scientific software. The first example consists of teaching scientific software development teams to utilize the practice of peer code review in order to increase the maintainability of their software through increasing the readability of the code and the preservation of knowledge across the team. The second example consists of developing and evaluating an semi-automated testing tool that scientific software developers can use to perform integration and regression testing on their software while requiring minimal extra effort on their software. In order to do create this tool, we utilized open-source scientific software projects as well as output from other scientific codes to create the testing suite. The use of open source projects is important because these projects have largely been accepted as providing useful tools for scientific software development, if a tool does not function with these open source projects then its usefulness will be limited.

In software that will be used for an extended period of time, the most expensive part of development is in the maintenance stage. In some cases, this stage can take up as much as 90% of the total effort devoted to a software project. While many scientific software projects are small projects that serve as more proof-of-concept than long term software development efforts, there are also a large number of projects, such as library development or searches for new materials, that are continually developed over many years. These projects, just as with traditional software engineering projects, will necessarily undergo a number of changes in their lifecycles. Because of this need for continued change, the developers will be required to perform maintenance tasks

المنارة للاستشارات
www.manaraa.com

on the software. Software engineers have determined that many factors contribute to the ease of maintaining software, including readability, preservation of knowledge across a team, and testing.

In addition to long-term projects, scientific software developers frequently explore many similar phenomena. The development process for these similar projects would be simplified if they could more easily reuse software. Many of the same factors that contribute to maintainability also contribute to reusability: readability, preservation of knowledge, and testing.

4.1.1 Subject Selection

The subjects for both of these examples are a team of scientific software developers at the Oak Ridge National Laboratory (ORNL). The team consists of four primary developers, all scientists. They develop materials simulations in C++ that are run both locally on the PCs of the team members and on the Titan Supercomputer. The software written by the team is primarily created in C++. We chose to work with this team because they were a team of scientists who were interested in partnering with software engineers to learn how to increase the quality of their software. Between the two examples one of the members of the team left and another member joined, but the overall expertise of the team remained the same.

This paper has three primary contributions.

1. An example of peer code review being utilized in a scientific software development project.
2. A checklist that can be used to help guide the peer code review process to identify common problems in scientific software development.
3. A tool that helps scientific software developers to perform integration and regression testing on their software.

The remainder of this paper will be structured as follows: Section 4.2 will present back-

المنارة للاستشارات
www.manaraa.com

ground information on software engineering for scientific software. Section 4.3 describes the Peer Code Review example. Section 4.4 describes the Testing example. Section 4.5 provides conclusions from across both examples.

4.2 Background

Because a scientific or engineering problem must be sufficiently complex to require the development of software, developers often need advanced technical training, most frequently a PhD, in the area to understand the needs of the problem. This situation differs from a traditional software development environment in which deep domain knowledge is not as strictly required. This requirement of detailed domain knowledge frequently means that a scientific software developer lacks the software development knowledge that a "traditional" software developer would have. In turn, various aspects of software quality may be lower. Software engineering provides techniques such as code reviews and testing that help developers. Code reviews help developers to reduce the number of defects in source code, increase the readability of code, and transfer knowledge between members of a software development team [13]. Integration testing helps developers find and fix problems that only become visible when multiple components of their software interact with one another [27]. Regression testing helps developers ensure that new development does not introduce errors in their software that was previously correct [2]. However, it appears that the prevalence of their use in scientific software is relatively low [36].

In addition to the software quality problems, scientists and engineers have a problem with productivity. According to Faulk et al., even though the speed of computers is rapidly increasing, it is becoming more difficult for scientists to actually do useful work. Faulk says that the reason for this situation is that "the dominant barriers to productivity improvement are in the software processes." In other words, the development approach that is primarily used in scientific software

المنارة للاستشارات
www.manaraa.com

contains bottlenecks. Faulk also claims that these bottlenecks cannot be removed "without fundamentally changing the way scientific software is developed" [18]. A major strength of software engineering is that it can increase productivity. Therefore, low productivity is another issue in which software engineering techniques can help scientific software developers.

Scientific software projects have a common set of characteristics which, according to Basili et al. [5], provide a source of knowledge that is essential to understand the claims made about the application of software engineering to scientific software projects.

First, many scientific software developers learn software development from other scientific software developers rather than via a formal software engineering education [11]. Unfortunately, the other scientific software developers also tend to lack formal software engineering training. Because scientific software developers do not have formal training, their ideas of what constitutes software engineering is limited. This lack of training means that they are likely unaware of techniques they could use that would allow them to have a much greater level of control over the quality of their code. Even when scientific software developers are familiar with certain software engineering techniques, they may not know how to properly apply them, leading them to decide that the cost of using software engineering techniques outweighs the benefits they provide.

Second, many of the software projects are not initially designed to be large, but do become large after initial trials prove successful [5]. Because the programs are not intended to be large, scientific software developers often do not take care to ensure that their code is easy to maintain. When scientific software developers have to later modify their code to add new features, these modifications require more effort than they should.

A final characteristic of scientific software is that it is generally used internally, that is either by its creator or by another member of the creator's research group [5]. Because the software is

المنارة للاستشارات
www.manaraa.com

used internally, the belief is that understandability by external developers is less important. As a result, the software is often difficult to read and poorly commented. These practices lead to software that is less maintainable, which is problematic if someone new joins the team or if one of the primary developers stops working on the code for some period and then returns to it.

It is important to understand that there is not one monolithic community of scientific software developers. According to Basili et al. [5], there are three primary variables that characterize the development of software for any individual researcher or group of researchers. The first variable is *team size*. In scientific software, the size of a team is usually either a single researcher who serves as his own developer or a large group. According to Basili, the large groups tend to consist of multiple groups that may not even be co-located. The second variable is the useful *lifetime* of the software. Software that is only expected to be executed once or twice does not require as much formal software engineering or need as much optimization as software that is going to be used multiple times, i.e. a scientific simulation or a scientific library. Additionally, when software is only executed once or twice, the effort required to optimize its performance can easily overwhelm the speedup it generates. The final variable is the *intended users* of the software. The users can be internal, external, or both. In the case of internal users, the developers do not tend to care as much about the quality of the user interface because they will be using the software themselves. When the software is going to be used by external users, the developers have to place more emphasis on the quality of the user interface as well as the qualities of readability and maintainability in order for others to be able to use it. Cases where both internal and external users are supported result in an additional layer of complication because multiple software versions must be maintained.

Traditional software development focuses on fulfilling the needs of a customer. This focus on the process has led software engineers to emphasize quality of the code itself. Scientific

المنارة للاستشارات
www.manaraa.com

software, on the other hand, exists to answer scientific or engineering questions that are difficult or impossible to answer experimentally due to constraints on time, expense, or the danger of performing the experiment. Because the most important goal for scientific software developers is the creation of new scientific knowledge, the relative emphasis scientific software developers place on various software quality attributes (i.e. correctness of code, maintainability, and reliability) has been historically lower than that given by traditional software developers [12]. Furthermore, there is no guarantee that software engineering techniques will work for scientific software development without modification. In fact, Segal, et al. suggest that software engineering techniques would have to be tailored for use in scientific software development [41].

4.3 Case Study 1: Peer Code Review

4.3.1 Problem and Research Objective

In order for a development team to maintain software, each part of the software must be readable to the entire team, as well as any new members that join the team after a portion of development is complete. We believe peer code reviews will increase the readability of scientific software because it has proven to do so in traditional software. Additionally, because at least two people are involved in the process, any issues that make sense to the developer but not a second observer can then be corrected.

Furthermore, one of the biggest challenges to reading a piece of code is the jarring transition between different coding standards, such as the use of camel-case or underscores to indicate spacing in variable names. In order to counter this challenge, a team should use one set of uniform standards.

Finally, we believe that the use of peer code reviews will significantly increase the preservation of knowledge across a scientific software development team as code reviews have been shown

المنارة للاستشارات
www.manaraa.com

to be an effective means of knowledge transmission in traditional software engineering development.

4.3.2 Background

Peer code reviews provide a wide range of benefits to software development teams. In traditional software engineering development, the use of peer code reviews has been found to have a wide range of benefits [13]. First, peer code reviews result in increased software quality [4, 17, 19, 42, 43, 45]. When someone that is not the primary developer looks at a piece of code, they are likely to find mistakes that the developer had been overlooking. Additionally, when a developer knows that someone else will be looking at his/her code s/he tends to make sure that it is both cleaner and better documented. Finally, the process of explaining how his/her software works can lead to a developer to realize that the software does not actually work as believed. Second, peer code review helps the transfer of knowledge across the members of a team [6, 7, 37, 45]. For example, a common problem in large programs is that each developer will focus on one part of the program to the point that there will be portions of the software that are only understood by one developer. If that developer leaves the company or is unavailable for some reason, another team member will have to learn that portion of the software on the fly instead of focusing on their own development tasks. Peer code review ensures that at least two members of the team are familiar with any given portion of the software, and likely more if the team members rotate review partners.

4.3.3 Study Design

In order to gain a greater understanding of the effectiveness of code reviews in scientific software development, we conducted a study with a team at ORNL. As the initial stage of performing this study we presented the team members with four code review options to determine the one that was most comfortable.

المنارة للاستشارات
www.manaraa.com

- *Formal Code Review:* at least four members of the team would come together to inspect a piece of code when the author believes it to be complete and ready for use: the author of the code, a "reader" who guides the examination of the code, a "moderator" who is responsible for organizing and reporting on the inspection, and some number of reviewers who critique the code. The code is inspected on the basis of determining the criteria or requirements that must be met to enter each process and the criteria or requirements which must be met to complete each process. These criteria are specified in a document prepared before the review session in order to more efficiently move through the process.
- *Over-the-shoulder Code Review:* the author of a piece of code that is believed to be complete and ready for use and one or two other members of the team meet at a single machine or a machine with a projector and go through the code together, focusing on the areas that jump out at them or anything the author is particularly concerned about. The author guides the discussion, explaining their reasons for making any decisions the reviewers have concerns about.
- *Remote Code Review:* utilizes the same process as the "over the shoulder" peer code review, but the people use screen sharing or a remote desktop program and communicate electronically.
- *Asynchronous Code Review:* the members of each team would be paired into groups of two or three that are familiar enough with each group member's portions of the codes to critique each others' code. When one of the members of the group feels a piece of code is complete and ready for use, they send it and the prior version of the code to the reviewers who can then use a diff to look at only the changes and respond to the author with their critiques.

المنارة للاستشارات
www.manaraa.com

The team members unanimously chose the "over the shoulder" peer code review for a number of reasons. First, they felt that the formal code review required much too large an investment of time and effort for the results they expected as most of the five-person team would be required to participate. Second, they felt that the asynchronous review would not provide a significant level of benefit over their current practices as the author would not be immediately available to explain the goals of the piece of code and answer any questions the reviewers had. Finally, the team saw no need for the "remote" peer code review as they were all located in the same building and could easily meet physically.

In this study one of the authors worked with the team to instruct them on how to perform peer code reviews and then observed the team as they performed peer code reviews. Once the teams were able to conduct the peer code reviews on their own, the author served as a scribe for the developers performing the code review. The scribe kept track of defects the team found, notable comments, the number of readability defects, and the number of functional defects. The author used this data to analyze the effectiveness of the peer code review process and extracted findings in Section 4.3.4

4.3.4 Results

During this study, we made four primary observations:

1. After participating in two review sessions, one as the reviewer and one as the reviewee, scientific developers were able to perform code reviews without further guidance by the author (a software engineer).
2. The use of code review motivated the team to develop and adopt a uniform coding standard.

المنارة للاستشارات
www.manaraa.com

3. Reviewers found defects that did not currently cause failures, but instead made the code less maintainable and sustainable.
4. The developers identified understandability problems about twice as often as they identified functionality defects.

Each of these observations show an effective improvement from the use of peer code reviews. Relative to Observation 1, these findings show that the practice of peer code reviews is easy enough for scientists to perform without extensive formal training. Because only a brief period of training is required, peer code reviews will be an easy process for scientists to adopt. Relative to Observation 2, it is notable that the developers adopted this practice on their own. While they were performing the review process, they found that their brains had to "shift gears" to understand each others' code without additional explanation. The use of a uniform coding style is considered a best practice in traditional software engineering because it allows the team to more easily understand the code they are working with. Observation 3 is particularly important because the developers specifically stated that they would not have found these problems with their current testing practices. Observation 4 supports the hypothesis that code reviews would greatly help the readability of the software.

4.3.5 Outcomes

The example of the development team at ORNL showed that peer code review was an effective practice and that the team members were able to perform the reviews without the ongoing help of a software engineer. Based on this success, we wanted to further increase the ability of scientific software developers to independently perform peer code reviews and eliminate the need for a software engineer to introduce the practice and provide structure to the review session. We determined that a checklist that could guide peer code review would help address this goal. The

المنارة للاستشارات
www.manaraa.com

most common obstacle to performing peer code review is focusing efforts on identifying the most important issues in a code document. A checklist would help developers make productive use of their review time, but is lightweight enough to not be a significant burden on their development effort. In order to develop these checklists, we performed two steps:

1. Survey the literature for papers that describe scientific software defects that appear to be common or significant and
2. Interview experienced scientific software developers to get their opinions on the types of items that have been most problematic in scientific software.

4.3.5.1 Literature Review

The literature review found that one of the primary difficulties facing scientific software development was that researchers do not, as a general rule, test their programs rigorously. As a result of this lack of rigorous testing, a number of authors speculated that the most dangerous defects were those that did not cause the program to break, which were obvious, but rather small defects that change the processing of data and result in significant differences in the output of the program [14, 32, 41]. In particular, scientific software developers tended to not perform integration testing, meaning that even if the individual pieces performed properly, the interaction of multiple functions could produce these small defects and go undetected until the developers thought the project was complete [15, 16, 36]. Furthermore, without continual integration testing, it is easy for these interactions to enter the portions of the software that are considered to be "good" and not become evident until developers try to add new functionality. The developers then frequently waste time trying to find bugs in their piece of the software that actually exist somewhere else [15].

Additionally, a number of papers found a frequent lack of documentation in scientific soft-

المنارة للاستشارات
www.manaraa.com

ware. Lack of documentation is a problem because, when scientists take working code from one software package to use on another, without documentation it is easy to implement the code incorrectly. In one case, a biologist had created a program to compare genomes to reconstruct evolutionary relationships in closely related organisms, but discovered that an independent group had taken his program and used it to look at organisms that were not as closely related as he had intended. After the independent team had published results, he found that his program did not produce valid results for organisms that were not within the bounds he had originally intended. Because he had not documented his original range, the team did not realize that they had received invalid results and published an incorrect conclusion. In another project, a team found that they had developed their software on a Linux operating system, but when they tried to using Apple computers they received noticeably different results because they had unintentionally tied the program to functions that operate differently in the Linux and Apple implementations. If this had been an external team that had attempted to utilize the project, the lack of documentation might well have resulted in the differences not being detected and resulted in another case of published invalid results [32]. Furthermore, the data formats frequently go undocumented, meaning that even though a development team makes their software available, future developers may not be able to understand how to use the program with their own data [24]. These findings suggest that the checklist needed questions about the following topics:

- If the developed function actually match the intended functionality,
- How the function interacts with the other functions in the program, and
- Whether the documentation for the function matches the indented functionality.

المنارة للاستشارات
www.manaraa.com

4.3.5.2 Interviews

The interviews were conducted at the SuperComputing 2014 Conference with attendees of the research presentations and workshops. The interviewees were approached randomly and asked if they would be willing to participate in a brief interview to help understand how scientific software developers could take advantage of code reviews. The interviewees were asked the following questions:

1. Do you usually develop code on your own or as part of a team?
2. If you develop code as part of a team, do your team members review each other's code regularly?
3. What are the most common types of problem that you find in your or your team's code?
4. What are the most important types of problems you find in your or your team's code?

We found that the developers primarily developed code as part of a team. Two-thirds of the developers only developed code as part of a team and the remainder developed code both on their own and as part of a team. Only one developer performed any type of code review, which was peer code review. The reasons the developer gave for using peer code review was that it was a quick process and s/he did not have the people or time to perform a more formal code review. The most important problems that people encountered were: everybody on their team used a different coding style, making it difficult to read the software as a whole, the ability of bugs to pass through testing unrecognized, and functions failing when they recieve unexpected input. These problems suggested that the checklist needed questions about coding standards and unexpected input.

المنارة للاستشارات
www.manaraa.com

4.3.5.3 Checklist

Based on the our findings from the ORNL study, the literature survey, and the interviews, we developed the following list of questions that should be helpful to aid scientific software developers in performing peer code reviews:

1. Does this piece of code fit our team's coding standards?
2. Is the intended functionality of this program documented?
3. Does the functionality match this documentation?
4. What is the expected range of input for this function?
5. What happens when we provide this function input that is on or beyond the edge of expected input?
6. What other functions use the information generated from this function?
7. Will this function work with our future plans for this software?

The first question is important because the most common problem the developers we interviewed encountered was that it was hard to understand the code produced by other members of their development team. Two of the developers interviewed specifically mentioned that everyone on their team, as well as people who had previously been on the team, used a different programming style. When these developers joined the team, this inconsistency caused them to spend much more time becoming familiar with the code than they initially expected. The remainder of the questions will need to be applied individually to each function that is examined. We also encountered a similar finding in our study at ORNL, when the developers realized that they would be able to better understand each other's code if they began using a uniform coding style. Questions two through seven are extremely important because it was these issues with the unintended input

المنارة للاستشارات
www.manaraa.com

that repeatedly appeared in the survey of the literature as causing the most important errors in the results of the programs. While the fifth question did not appear in the survey of the literature, it covers a situation that arose during the study at ORNL. One of the team members had written a piece of software that functioned correctly, but when the team leader came across it in the review he realized that a piece of functionality he had intended to add in the future would be much simpler to implement with a few minor changes to the function they were reviewing.

4.3.5.4 Future Work

We plan to further evaluate this checklist to determine its current effectiveness and identify potential improvements. In order to do this evaluation, we plan to conduct a formal case study with a scientific software development team that does not currently use peer code review. We will provide the team with the checklist as well as any training they require to understand the items on the checklist. As the team uses the checklist to guide their code reviews, we will ask the team to record the type of each defect they find as well as which checklist question (if any) led to the detection of that defect. Additionally, we will periodically conduct surveys and interviews to collect qualitative analysis as to the usefulness of the checklist.

4.4 Case Study 2: Testing

4.4.1 Problem and Research Objective

One of the qualities required to re-use existing software is that it be correct. The effectiveness of testing techniques, which help ensure correctness, has been repeatedly identified as one of the largest problems facing scientific software development [1, 15, 16, 28, 33, 34, 36, 38, 39, 41]. However, it has also been repeatedly identified that the testing performed by scientific software developers is either of limited effectiveness or executed poorly [1, 15, 16, 28, 33, 34, 36, 38, 39, 41] In order to address this discrepancy, we formed the following hypothesis: *The use of regression*

المنارة للاستشارات
www.manaraa.com

and integration testing in concert will provide a means for scientific software developers to show that their software is "more correct." This hypothesis is based upon the belief that these techniques allow developers to show that they have not made anything worse by comparing to a base model that is accepted as accurate.

4.4.2 Background

Regression testing serves to detect any failures introduced by new development on software that has previously been determined to be correct. Regression testing can be performed at both the system and the unit level. When tests are specified at the system level, it is possible for developers to get more test-case reuse when they begin creating unit tests [30]. When the regression tests are applied at the unit level, however, developers are able to detect defects earlier when they are less expensive to repair [22, 31]. Regression testing has been found to be much simpler to perform when joined with tools that help automate the repeated testing [35, 46].

Integration testing helps developers detect failures that arise when multiple pieces of software interact. Integration testing focuses on the communication between different components of the overall software project [25, 27]. Integration testing has been incorporated into ISO, CMMI, and SPICE Automotive standards [21, 26]. As mentioned in Section 4.3.5.1, these failures are some of the most important problems facing scientific software developers. As with regression testing, integration testing is greatly aided by automation [23, 29, 40].

4.4.3 Development of TestSci

As we showed in Section 4.4.2, regression and integration testing have both been found to address important defects that are introduced by the change or addition of code to existing software. However, both testing techniques require a large amount of repetitive, time-consuming work which makes them less appealing to developers who are not familiar with their benefits. Fortunately, both

المنارة للاستشارات
www.manaraa.com

practices have been reported in the literature as being greatly augmented by automation. Because we wanted to provide as much benefit to our scientific partners as possible without requiring a large amount of additional work, we decided that it was necessary to produce a tool that would allow the testing to be performed overnight and leave the development team with a series of simple log files that they could quickly work through when they began work the next day.

In order to develop TestSci, we partnered with a team at ORNL. First, we interviewed the team to determine what problems they encountered most frequently in their software development. During this interview, we found that one of the major issues they encountered was new functionality failing to interact correctly with existing functionality. Based on this finding, we determined that regression testing was one of the techniques needed to help the team address their problems. Additionally, the project has been under development without true unit testing for many years, so we determined that the system level regression testing would be most appropriate for the development team. After digging deeper, the team also reported that they had issues arise when they were attempting to incorporate pieces of software that had been written by different developers. This problem suggests that integration testing was also an appropriate practice for the team to adopt. As both regression and integration testing have been shown to be simplified by the use of an automated testing tool, we decided to create TestSci, a tool that can be used to automatically test a number of configurations of a scientific program.

TestSci consists of two parts. The first part of TestSci is a Bash script that a develeloper can use to automate the process of running multiple configurations of the same project. This script allows the developer to set up each configuration he/she wants to test ahead of time and then run the script. With no further interaction, the script runs each configuration, passes the results to the testing portion of TestSci, and stores the results from TestSci in a separate folder for each

المنارة للاستشارات
www.manaraa.com

Figure 4.1: Error output from TestSci

```
0 errors in File: info_evec_out

2 errors in File: k.out
Error: line 4, item 1 in file .//k.out
−5082.166495463603 != −5081.166495463603
Error: line 7, item 1 in file .//k.out
−5082.166529648014 != −5083.166529648014

0 errors in File: w_fe2.0

1 errors in File: w_fe2.1
Error: line 1541, item 3 in file .//w_fe2.1
0.8778531269700D−13 != 0.8775531269700D−13
```

configuration. The testing portion of TestSci provides two major functionalities. The tool compares the output from a run of the software it is being used to test to known good data. If a difference in the data exceeds a threshold set in the configuration file for TestSci, then the file name, the line number, the correct data, the new data, and the difference between the two will be saved to a log file in order to help the user determine what went wrong. Figure 4.1 provides an example of this output. After the error check, depending on whether or not any errors were detected, TestSci does one of two things. If there were no errors detected, then TestSci saves a copy of that version of the software. However, if there are errors detected, then TestSci compares the current version of the software to the most recent saved version and gives the user a detailed report of changes, including the file(s) the changes are in, the line numbers of the changes, and a comparison of the original code to the new code. Figure 4.2 illustrates an example of this comparison. The primary benefit of using TestSci is that it provides the user with information about errors and code changes that they can then use to more quickly identify the cause of the errors. When the user takes advantage of

المنارة للاستشارات
www.manaraa.com

Figure 4.2: Code Change output from TestSci

```
--- C:/Fe2/numpy-1.9.1/numpy/polynomial/chebyshev.py        3
+++ C:/Fe2/numpy-1.9.2/numpy/polynomial/chebyshev.py
@@ -280,7 +280,7 @@

     """
     n = len(zs)//2
-    ns = np.array([-1, 0, 1], Ftype=zs.dtype)
+    ns = np.array([-1, 0, 1], dtype=zs.dtype)
     zs *= np.arange(-n, n+1)*2
     d, r = _zseries_div(zs, ns)
     return d


--- C:/Fe2/numpy-1.9.1/numpy/polynomial/hermite_e.py        3
+++ C:/Fe2/numpy-1.9.2/numpy/polynomial/hermite_e.py
@@ -291,7 +291,7 @@
         [roots] = pu.as_series([roots], trim=False)
         roots.sort()
         p = [hermeline(-r, 1) for r in roots]
-        n = len(p+1)
+        n = len(p)
         while n > 1:
             m, r = divmod(n, 2)
             tmp = [hermemul(p[i], p[i+m]) for i in range(m)]
@@ -346,7 +346,7 @@
         ret = c1
     else:
         c2[:c1.size] += c1
-        ret = c2 -1
+        ret = c2
     return pu.trimseq(ret)
```


المنارة للاستشارات
www.manaraa.com

this information, they can spend less time trying to find the errors manually and more time fixing errors and working on new features.

4.4.4 Proof-of-Concept Example

In order to show that TestSci was effective at its goal of aiding the developer in performing integration and regression testing, we used output given to us by our partners to provide a real-world test of TestSci's ability to decrease the time that it takes to detect errors in the output of changed code. In order to perform this test, we seeded the output files with 3 errors and used TestSci to generate an error log. The original output consisted of 3096 lines of text, while the error log from TestSci is completely contained in figure 4.1 In this somewhat common case, that errors would only appear on a few lines of the output, the reviewer would have had to work through 0.45% as much text to use the errors in the log file than if they were manually inspecting the output files.

To test the change-detection functionality of TestSci, we similarly seeded the polynomial library of numpy with errors. In this case, a manual search for the changes would require going through 13,083 lines of code while the change log from TestSci is contained in Figure 4.2. The developers would then have to work through 0.27% as much text to find the changes than if they were manually inspecting the code files. While this is a best-case scenario, regular use of the tool would keep the change-set relatively small as compared to the overall size of the software.

The much lower amount of information to process shows that using TestSci will prove to be much more efficient than the manual process previously used by our partners. When used on a regular basis, TestSci will provide the developers with a concise report containing a parsed version of the output they already had that is much easier for them to use to quickly detect the presence of errors. The addition of the code change information will provide two functional use case options,

المنارة للاستشارات
www.manaraa.com

depending on the team's development methodology. First, it allows a single developer to easily go through the changes made by the other developers on the team in order to identify which change is responsible for the error in output. Second, the team leader can use the code change information to divide this inspection among the members of the team more evenly and guide the team members to the locations that the errors could have entered the software. The primary use of TestSci is to aid the developer in performing integration testing by simplifying the process of finding the errors introduced by the interaction between otherwise tested components. TestSci also provides regression testing support by automatically saving the version of the code that produced correct results. The change analysis created based on this information helps the developer identify whether the problem was introduced by new functionality or a change in existing functionality.

4.4.5 Future Work

In addition to this proof-of-concept, we plan to validate in the real world context of our partners at ORNL. In order to perform this further validation, we created a version of TestSci with a script tailored to their project and provided it for their use. Once they have used TestSci for an extended period of time (1 or 2 months), we will conduct another interview to gather the following data:

1. How much extra work was required to use TestSci in their normal process of software development,
2. How much TestSci decreased the time spent analyzing the data output,
3. How much TestSci helped the process of identifying changed code that modified the original functionality, and

www.manaraa.com

4. What additional functionality would help TestSci be more useful for their development process.

Based on this additional input, we plan to determine what did and did not work with the current iteration of TestSci and improve it for release as an open-source tool for scientific software developers.

4.5 Summary and Future Work

In this paper we presented a pair of examples covering the process of peer code review and the development and evaluation of TestSci, a semi-automated testing tool to aid scientific software developers in their development process. While addressing the peer code review example, we presented findings from teaching a team of scientific software developers to take advantage of the peer code review process. We found that scientific software developers were able to perform peer code review on their own after two sessions with the aid of a software engineer, develop a uniform coding standards, find defects that would cause errors in the future, and find a significant number of readability defects in addition to functional defects. Based on these findings, as well as a literature survey and interviews with developers, we presented a checklist that can be used to help scientific software developers learn to perform effective peer code review without the assistance of software engineers.

We presented a preliminary evaluation of the effectiveness of TestSci. We found that the output produced by TestSci was much easier to navigate and understand than the raw data that had been previously used by our partners. A potential threat to the validity of this evaluation is that the evaluation was performed by one of the authors and only looked at the quantitative data of lines of output. Because the errors were injected, we cannot say how much effort was saved by

www.manaraa.com

the use of the tool. In order to address this threat we plan to have the team we partnered with to evaluate TestSci independently in the course of their normal development, improve the tool based on their findings, and release it open-sourced to the software engineering and scientific software development communities.

للاستشارات المنارة
www.manaraa.com

REFERENCES

[1] K. Ackroyd, S. Kinder, G. Mant, M. Miller, C. Ramsdale, and P. Stephenson, "Scientific software development at a research facility," *IEEE Software*, vol. 25, no. 4, pp. 44–51, 2008.

[2] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental regression testing," in *Proceedings of the Conference on Software Maintenance*, 1993, pp. 348–357.

[3] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 712–721.

[4] R. Ballantyne, K. Hughes, and A. Mylonas, "Developing procedures for implementing peer assessment in large classes using an action research process," *Assessment & Evaluation in Higher Education*, vol. 27, no. 5, pp. 427–441, 2002.

[5] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz, "Understanding the high-performance-computing community: A software engineer's perspective," *IEEE Software*, vol. 25, no. 4, pp. 29–36, Jul. 2008.

[6] F. Belli and R. Crisan, "Towards automation of checklist-based code-reviews," in *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, Oct. 1996, pp. 24–33.

[7] A. Bhalerao and A. Ward, "Towards electronically assisted peer assessment : A case study," *ALT-J : research in learning technology*, vol. Volume 9, no. 1, pp. 26–37, 2001.

[8] B. W. Boehm, *Software Engineering Economics*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[9] M. Broy, F. Deissenboeck, and M. Pizka, "Demystifying maintainability," in *Proceedings of the 2006 International Workshop on Software Quality*, 2006, pp. 21–26.

[10] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008, pp. 121–130.

المنارة للاستشارات
www.manaraa.com

[11] J. C. Carver, “Report from the second international workshop on software engineering for computational science and engineering,” *Computing in Science and Engineering*, vol. 11, no. 6, pp. 14–19, 2009.

[12] ——, “SE-CSE 2008: The first international workshop on software engineering for computational science and engineering,” in *Companion Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 1071–1072.

[13] L. Colen, “Code reviews,” *Linux Journal*, no. 81, Jan. 2001.

[14] P. Dubois, “Testing scientific programs,” *Computing in Science and Engineering*, vol. 14, no. 4, pp. 69–73, Jul. 2012.

[15] ——, “Maintaining correctness in scientific programs,” *Computing in Science and Engineering*, vol. 7, no. 3, pp. 80–85, 2005.

[16] S. M. Easterbrook and T. C. Johns, “Engineering the software for understanding climate change,” *Computing in Science and Engineering*, vol. 11, no. 6, pp. 65–74, 2009.

[17] S. Fallows and B. Chandramohan, “Multiple approaches to assessment: Reflections on use of tutor, peer and self-assessment,” *Teaching in Higher Education*, vol. 6, no. 2, pp. 229–246, 2001.

[18] S. Faulk, E. Loh, M. L. Vanter, S. Squires, and L. G. Votta, “Scientific computing’s productivity gridlock: How software engineering can help,” *Computing in Science and Engineering*, vol. 11, no. 6, pp. 30–39, 2009.

[19] E. F. Gehringer, D. D. Chinn, M. A. Pérez-Quiñones, and M. A. Ardis, “Using peer review in teaching computing,” in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 2005, pp. 321–322.

[20] N. S. Gill, “Factors affecting effective software quality management revisited,” *SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–4, Mar. 2005.

[21] A. S. I. Group, “Automotive SPICE process assessment model,” Automotive Special Interest Group, Tech. Rep., 2007.

[22] R. A. Haraty, N. Mansour, and B. Daou, “Regression testing of database applications,” in *Proceedings of the 2001 ACM Symposium on Applied Computing*, 2001, pp. 285–289.

[23] R. Hewett and P. Kijsanayothin, “Automated test order generation for software component integration testing,” in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 211–220.

المنارة للاستشارات
www.manaraa.com

[24] K. Hinsen, "Software development for reproducible research," *Computing in Science and Engineering*, vol. 15, no. 4, pp. 60–63, 2013.

[25] D. Hura and M. Dimmich, "A method facilitating integration testing of embedded software," in *Proceedings of the Ninth International Workshop on Dynamic Analysis*, 2011, pp. 7–11.

[26] C. M. S. E. Institute, "CMMI for development," Carnegie Mellon Software Engineering Institute, Tech. Rep., 2006.

[27] P. C. Jorgensen and C. Erickson, "Object-oriented integration testing," *Communications of the ACM*, vol. 37, no. 9, pp. 30–38, Sep. 1994.

[28] K. Karhu, T. Repo, O. Taipale, and K. Smolander, "Empirical observations on software testing automation," in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, 2009, pp. 201–209.

[29] T. M. King, A. S. Ganti, and D. Froslie, "Enabling automated integration testing of cloud application services in virtualized environments," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, 2011, pp. 120–132.

[30] H. L. L. White, "On the edge. regression testability," *IEEE Micro*, vol. 12, no. 2, pp. 81–84, Mar. 1992.

[31] A. McCarthy, "Unit and regression testing," *Dr. Dobb's Journal*, pp. 18–20, 82, & 84, Feb. 1997.

[32] Z. Merali, "Computational science: Error, why scientific computing does not compute," *Nature*, vol. 467, no. 7317, pp. 775–777, 2010.

[33] M. Miic, M. Tomaevic, and I. Bethune, "Automated multi-platform testing and code coverage analysis of the CP2K application," in *Seventh International Conference on Software Testing, Verification and Validation*, Mar. 2014, pp. 95–98.

[34] C. Morris and J. Segal, "Some challenges facing scientific software developers: The case of molecular biology," in *Fifth IEEE International Conference on e-Science*, 2009, pp. 216–222.

[35] M. H. Netkow and D. Brylow, "Xest: An automated framework for regression testing of embedded software," in *Proceedings of the 2010 Workshop on Embedded Systems Education*, 2010, pp. 71–78.

المنارة للاستشارات
www.manaraa.com

[36] L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana, “A survey of scientific software development,” in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 12:1–12:10.

[37] J.-S. Oh and H.-J. Choi, “A reflective practice of automated and manual code reviews for a studio project,” in *Proceedings of the Fourth Annual ACIS International Conference on Computer and Information Science*, 2005, pp. 37–42.

[38] Y. Pouillon, J. Beuken, T. Deutsch, M. Torrent, and X. Gonze, “Organizing software growth and distributed development: The case of Abinit,” *Computing in Science and Engineering*, vol. 12, no. 1, pp. 62–69, (Jan-Feb) 2011 2011.

[39] H. Remmel, B. Paech, C. Engwer, and P. Bastian, “Design and rationale of a quality assurance process for a scientific framework,” in *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering*, May 2013, pp. 58–67.

[40] F. Saglietti and F. Pinte, “Automated unit and integration testing for component-based software systems,” in *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems*, 2010, pp. 5:1–5:6.

[41] J. Segal, “Some challenges facing software engineers developing software for scientists,” in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009, pp. 9–14.

[42] E. Silva and D. Moreira, “Webcom: A tool to use peer review to improve student interaction,” *Journal of Educational Resources in Computing*, vol. 3, no. 1, Mar. 2003.

[43] D. Sluijsmans, F. Dochy, and G. Moerkerke, “Creating a learning environment by using self-, peer- and co-assessment,” *Learning Environments Research*, vol. 1, no. 3, pp. 293–319, 1998.

[44] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, “The structure and value of modularity in software design,” *SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 99–108, Sep. 2001.

[45] D. A. Trytten, “A design for team peer code review,” *SIGCSE Bulletin*, vol. 37, no. 1, pp. 455–459, Feb. 2005.

[46] N. J. Wahl, “An overview of regression testing,” *SIGSOFT Software Engineering Notes*, vol. 24, no. 1, pp. 69–73, Jan. 1999.

المنارة للاستشارات
www.manaraa.com

()

المنارة للاستشارات
www.manaraa.com

Chapter 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusion

The dissertation shows that, while scientific software developers recognize they would benefit from using software engineering practices, practices that support verification & validation and testing have not been widely adopted. This conclusion is important because these areas have been repeatedly identified by both scientists and software engineers as some of the most difficult challenges facing scientific software development. Additionally, this dissertation showed that many scientific software developers have adopted practices that approximate the agile development approach, even when they have no formal training in that approach. Furthermore, this dissertation shows that scientific software developers were generally unable to evaluate their overall knowledge of software engineering as shown by their lack of knowledge of specific software engineering practices. In particular, scientists are unfamiliar with the testing practices that would help address the challenges of verification & validation. Finally, the dissertation shows that the software engineering practices of peer code reviews, integration testing, and regression testing are effective at addressing the issues of maintainability and readability in scientific software development.

5.2 Contributions

The primary contributions of this dissertation follow. First, I found that, while scientific software developers recognize they would benefit from using software engineering practices, practices that support verification & validation and testing have not been widely adopted. However,

المنارة للاستشارات
www.manaraa.com

the difficulties of validating, verifying, and testing their software are widely viewed as one of the most important problems facing scientific software developers. Additionally, I created a list of the software engineering practices used by scientific software developers, provided an analysis of the effectiveness of those practices as well as an analysis of the evidence used to evaluate this effectiveness. Furthermore, I found that scientific software developers were generally unable to evaluate their overall knowledge of software engineering as shown by their knowledge of specific software engineering practices. In particular, I found that scientists were not knowledgeable about a variety of common software testing practices. This observation is important because it means that scientific software developers are not familiar with the practices that would serve to solve the problems with verification & validation shown above. Finally, I showed that the software engineering practices of peer code reviews, integration testing, and regression testing are effective at addressing the issues of maintainability and readability in scientific software development.

5.3 Future Work

In addition to the work performed in this dissertation, I have two more studies planned. The first study is to provide the checklist generated in the work described in Chapter 4 to a group of scientific software developers that is unfamiliar with the process of performing peer code reviews. I will have the team use the checklist to guide their code review process for two months and then have the team members complete a survey to evaluate the effectiveness of the code review process. Additionally, I will have the team participate in an interview to determine how the checklist could be improved and perform another iteration of the survey. The second study is to conduct a more formal user study to validate the SciTest tool also described in Chapter 4. Based on the results of this user study, I plan to revise SciTest and release it as an open-source project to benefit the scientific software development community.

المنارة للاستشارات
www.manaraa.com

5.4 Publications

The major work for this dissertation is being published in journals as follows: The literature review covered in Chapter 2, titled "Claims About the Use of Software Engineering Practices in Science: A Systematic Literature Review," has been submitted to *Information Software & Technology* and was returned for a major revision, to be submitted shortly after completion of the dissertation. The paper based on the surveys covered in Chapter 3, titled "What Scientists and Engineers Know About Software Engineering: A Survey," is under review at the *Empirical Software Engineering Journal*. The paper based on the case studies presented in Chapter 4, will be submitted to the *Empirical Software Engineering Journal*.

The work covered by this dissertation has also been published in the following venues.

1. Self-Perceptions about Software Engineering: A Survey of Scientists and Engineers in the *Computing in Science and Engineering* magazine[6],
2. The Relationship between Development Problems and Use of Software Engineering Practices in Computational Science & Engineering: A Survey at the *First Workshop on Maintainable software Practices in e-Science*[12], and
3. What Software Engineering Can Do for Computational Science and Engineering at the *IEEE Symposium on Visual Languages and Human-Centric Computing*[11].

www.manaraa.com

REFERENCES

[1] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 712–721.

[2] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz, "Understanding the high-performance-computing community: A software engineer's perspective," *IEEE Software*, vol. 25, no. 4, pp. 29–36, Jul. 2008.

[3] B. W. Boehm, *Software Engineering Economics*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[4] M. Broy, F. Deissenboeck, and M. Pizka, "Demystifying maintainability," in *Proceedings of the 2006 International Workshop on Software Quality*, 2006, pp. 21–26.

[5] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008, pp. 121–130.

[6] J. Carver, D. Heaton, L. Hochstein, and R. Bartlett, "Self-perceptions about software engineering: A survey of scientists and engineers," *Computing in Science Engineering*, vol. 15, no. 1, pp. 7–11, Jan. 2013.

[7] J. C. Carver, "Report from the second international workshop on software engineering for computational science and engineering," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 14–19, 2009.

[8] ——, "SE-CSE 2008: The first international workshop on software engineering for computational science and engineering," in *Companion Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 1071–1072.

[9] S. Faulk, E. Loh, M. L. Vanter, S. Squires, and L. G. Votta, "Scientific computing's productivity gridlock: How software engineering can help," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 30–39, 2009.

[10] N. S. Gill, "Factors affecting effective software quality management revisited," *SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–4, Mar. 2005.

المنارة للاستشارات
www.manaraa.com

[11] D. Heaton, "What software engineering can do for computational science and engineering," in *Proccedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing*, Sep. 2012, pp. 225–226.

[12] D. Heaton, J. Carver, R. Bartlett, K. Oakes, and L. Hochstein, "The relationship between development problems and use of software engineering practices in computational science & engineering: A survey," in *Proccedings of the 1st International workshop on Maintainable Software Practices in e-Science*, May 2012.

[13] J. Segal, "Some challenges facing software engineers developing software for scientists," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, 2009, pp. 9–14.

[14] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," *SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 99–108, Sep. 2001.

المنارة للاستشارات
www.manaraa.com

Appendices

المنارة للاستشارات
www.manaraa.com

Office for Research

Institutional Review Board for the
Protection of Human Subjects

THE UNIVERSITY OF
ALABAMA
RESEARCH

October 24, 2014

Dustin Heaton
Department of Computer Science
College of Engineering
The University of Alabama
Box 870290

Re: IRB # EX-14-CM-117 "Interview of Scientific Software Developers Regarding Peer Code Review"

Dear Mr. Heaton:

The University of Alabama Institutional Review Board has granted approval for your proposed research.

Your protocol has been given exempt approval according to 45 CFR part 46.101(b)(2) as outlined below:

(2) Research involving the use of educational tests (cognitive, diagnostic, aptitude, achievement), survey procedures, interview procedures or observation of public behavior, unless:
(i) information obtained is recorded in such a manner that human subjects can be identified, directly or through identifiers linked to the subjects; and (ii) any disclosure of the human subjects' responses outside the research could reasonably place the subjects at risk of criminal or civil liability or be damaging to the subjects' financial standing, employability, or reputation.

Your application will expire on October 23, 2015. If your research will continue beyond this date, complete the relevant portions of Continuing Review and Closure From. If you wish to modify the application, complete the Modification of an Approved Protocol Form. When the study closes, complete the appropriate portions of FORM: Continuing Review and Closure.

Should you need to submit any further correspondence regarding this proposal, please include the assigned IRB application number.

Good luck with your research.

Sincerely,

Stuart Usdan, Ph.D.
Chair, Non-Medical Institutional Review Board
The University of Alabama

358 Rose Administration Building
Box 870127
Tuscaloosa, Alabama 35487-0127
(205) 348-8461
FAX (205) 348-7189
TOLL FREE (877) 820-3066

IRB Project #: EX-14-CM-117

OCT 09 2014 AM 9:51

UNIVERSITY OF ALABAMA
INSTITUTIONAL REVIEW BOARD FOR THE PROTECTION OF HUMAN SUBJECTS
REQUEST FOR APPROVAL OF RESEARCH INVOLVING HUMAN SUBJECTS

I. Identifying information

	Principal Investigator	Second Investigator	Third Investigator
Names:	Dustin Heaton	Dr. Jeffrey C. Carver	
Department:	Computer Science	Computer Science	
College:	Engineering	Engineering	
University:	Alabama	Alabama	
Address:	Box 870290	Box 870290	
Telephone:		8-9829	
FAX:		8-0219	
E-mail:	dwheaton@crimson.ua.edu	carver@cs.ua.edu	

Title of Research Project: Interview of Scientific Software developers regarding peer code review

Date Submitted: October 7, 2014
Funding Source:

Type of Proposal ☒ New | ☐ Revision | ☐ Renewal Please attach a renewal application | ☐ Completed | ☐ Exempt
Please attach a continuing review of studies form
Please enter the original IRB # at the top of the page

UA faculty or staff member signature: ____________

II. NOTIFICATION OF IRB ACTION (to be completed by IRB):
Type of Review: ______ Full board _____ Expedited

IRB Action:
___ Rejected Date: ________
___ Tabled Pending Revisions Date: ________
___ Approved Pending Revisions Date: ________
✓ Approved-this proposal complies with University and federal regulations for the protection of human subjects.

Approval is effective until the following date: 10-23-15 F.S.

Items approved: ___ Research protocol (dated________)
___ Informed consent (dated________)
___ Recruitment materials (dated________)
___ Other (dated________)

Approval signature ____________ Date 10-24-14

UNIVERSITY OF ALABAMA

Informed Consent for a Research Study

You are being asked to take part in a research study. This study is called Interview of Scientific Software Developers Regarding Peer Code Review. The study is being done by Dr. Jeffrey Carver and Dustin Heaton of the University of Alabama

What is this study about?

This study will help us understand how Computational Science and Engineering (CSE) developers can benefit from a structured peer review process.

Why is this study important--What good will the results do?

This study will help us better understand the review process of CSE developers. It will also provide information that we will use to develop checklists that can be used to help guide future reviews

Why have I been asked to take part in this study?

You have been asked to be in this study because you are an attendee of the SuperComputing Conference.

How many people besides me will be in this study?

Approximately 100 people will participate in the study.

What will I be asked to do in this study?

If you decide to be in this study, you will be asked to complete an interview.

How much time will I spend being in this study?

Approximately 10 minutes.

Will I be paid for being in this study?

No.

Will being in this study cost me anything?

There will be no cost to you.

Can the researcher take me out of this study?

The researcher may take you out of the study if your interview is incomplete.

What are the benefits (good things) that may happen to me if I am in this study?

There are no direct benefits to you from being in this study.

What are the benefits to scientists or society?

UA IRB Approved Document
Approval date: 10-24-14
Expiration date: 10-23-15

This study will help the researchers understand current computational science software review practices and the areas that need the most support to ensure software quality.

What are the risks (dangers or harm) to me if I am in this study?

There are no risks to participating in this study. All information will be anonymous.

How will my confidentiality (privacy) be protected? What will happen to the information the study keeps on me?

The interviews will be anonymous.

What are the alternatives to being in this study? Do I have other choices?

The alternative/other choice is not to participate.

What are my rights as a participant?

Taking part in this study is voluntary. You may choose not to take part at all. If you start the study, you can stop at any time. Leaving the study will not result in any penalty or loss of any benefits you would otherwise receive.

Who do I call if I have questions or problems?

If you have questions about the study, please contact Dr. Carver at (205)-348-9828 or carver@cs.ua.edu. If you have any questions about your rights as a research participant you may contact Ms. Tanta Myles, The University of Alabama Research Compliance Officer, at (205)-348-8461 or 1-877-820-3066.

If you have read this consent form, had the study adequately explained to you, understand what you are being asked to do, and freely agree to take part in the interview, please give consent to be interviewed and have your responses used.

UA IRB Approved Document
Approval date: 10-24-14
Expiration date: 10-23-15

Office for Research

Institutional Review Board for the
Protection of Human Subjects

April 23, 2015

Dustin Heaton
Department of Computer Science
College of Engineering
The University of Alabama
Box 870290

Re: IRB # EX-15-CM-066 "Automated Testing for Computational Science and Engineering"

Dear Mr. Heaton:

The University of Alabama Institutional Review Board has granted approval for your proposed research.

Your protocol has been given exempt approval according to 45 CFR part 46.101(b)(2) as outlined below:

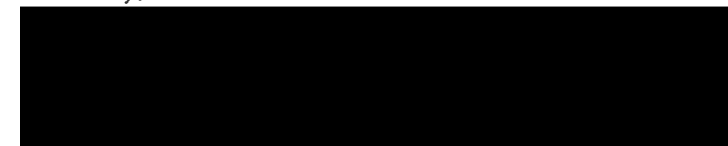
(2) Research involving the use of educational tests (cognitive, diagnostic, aptitude, achievement), survey procedures, interview procedures or observation of public behavior, unless:
(i) information obtained is recorded in such a manner that human subjects can be identified, directly or through identifiers linked to the subjects; and (ii) any disclosure of the human subjects' responses outside the research could reasonably place the subjects at risk of criminal or civil liability or be damaging to the subjects' financial standing, employability, or reputation.

Your application will expire on April 22, 2016. If your research will continue beyond this date, complete the relevant portions of Continuing Review and Closure From. If you wish to modify the application, complete the Modification of an Approved Protocol Form. When the study closes, complete the appropriate portions of FORM: Continuing Review and Closure.

Should you need to submit any further correspondence regarding this proposal, please include the assigned IRB application number.

Good luck with your research.

Sincerely,



Director & Research Compliance Officer
Office for Research Compliance

358 Rose Administration Building
Box 870127
Tuscaloosa, Alabama 35487-0127
(205) 348-8461
FAX (205) 348-7189
TOLL FREE (877) 820-3066

IRB Project #: EX-15-CM-066

UNIVERSITY OF ALABAMA
INSTITUTIONAL REVIEW BOARD FOR THE PROTECTION OF HUMAN SUBJECTS
REQUEST FOR APPROVAL OF RESEARCH INVOLVING HUMAN SUBJECTS

I. Identifying information

	Principal Investigator	Second Investigator	Third Investigator
Names:	Dustin Heaton	Dr. Jeffrey C. Carver	
Department:	Computer Science	Computer Science	
College:	Engineering	Engineering	
University:	Alabama	Alabama	
Address:	Box 870290	Box 870290	
Telephone:		8-9829	
FAX:		8-0219	
E-mail:	dwheaton@crimson.ua.edu	carver@cs.ua.edu	

Title of Research Project: Automated Testing for Computational Science and Engineering

Date Submitted: April 14, 2015
Funding Source: National Science Foundation
OSP#:14-0526

Type of Proposal	☒ New	☐ Revision	☐ Renewal Please attach a renewal application	☐ Completed	☑ Exempt
			Please attach a continuing review of studies form		
		Please enter the original IRB # at the top of the page			

UA faculty or staff member signature: ______________________

II. NOTIFICATION OF IRB ACTION (to be completed by IRB):
Type of Review: ______ Full board _____ Expedited

IRB Action:
___ Rejected Date: _________
___ Tabled Pending Revisions Date: _________
___ Approved Pending Revisions Date: _________
✓ Approved-this proposal complies with University and federal regulations for the protection of human subjects.

Approval is effective until the following date: 4-22-16
Items approved: ___ Research protocol (dated____________)
___ Informed consent (dated____________)
___ Recruitment materials (dated____________)
___ Other (dated____________)

Approval signature ______________________ Date 4/23/2015

UNIVERSITY OF ALABAMA

Informed Consent for a Research Study

You are being asked to take part in a research study. This study is called Automated Testing for Computational Science and Engineering. The study is being done by Dr. Jeffrey Carver and Dustin Heaton of the University of Alabama

What is this study about?

This study will help us understand how Computational Science and Engineering (CSE) developers take advantage of automated unit, regression, and integration testing.

Why is this study important--What good will the results do?

This study will help us better understand the review process of CSE developers. It will also provide information that we will use to develop checklists that can be used to help guide future reviews

Why have I been asked to take part in this study?

You have been asked to be in this study because you are a member of a software development team at Oak Ridge National Laboratory.

How many people besides me will be in this study?

Approximately 5 people will participate in the study.

What will I be asked to do in this study?

If you decide to be in this study, you will be asked to complete an interview to determine the appropriate "unit" of code to test, utilize the testing tool generated by Dustin Heaton, and complete a survey and interview to give your opinion of the usefulness of the testing techniques.

How much time will I spend being in this study?

Approximately 10 minutes.

Will I be paid for being in this study?

No.

Will being in this study cost me anything?

There will be no cost to you.

Can the researcher take me out of this study?

The researcher may take you out of the study if your interview is incomplete.

What are the benefits (good things) that may happen to me if I am in this study?

There are no direct benefits to you from being in this study.

UNIVERSITY OF ALABAMA IRB
CONSENT FORM APPROVED: 4-23-15
EXPIRATION DATE: 4-22-16

What are the benefits to scientists or society?

This study will help the researchers understand what tool support is needed to support testing in the computational science and engineering domain.

What are the risks (dangers or harm) to me if I am in this study?

There are no risks to participating in this study. All information will be anonymous.

How will my confidentiality (privacy) be protected? What will happen to the information the study keeps on me?

The interviews and survey will be anonymous.

What are the alternatives to being in this study? Do I have other choices?

The alternative/other choice is not to participate.

What are my rights as a participant?

Taking part in this study is voluntary. You may choose not to take part at all. If you start the study, you can stop at any time. Leaving the study will not result in any penalty or loss of any benefits you would otherwise receive.

Who do I call if I have questions or problems?

If you have questions about the study, please contact Dr. Carver at (205)-348-9828 or carver@cs.ua.edu. If you have any questions about your rights as a research participant you may contact Ms. Tanta Myles, The University of Alabama Research Compliance Officer, at (205)-348-8461 or 1-877-820-3066.

If you have read this consent form, had the study adequately explained to you, understand what you are being asked to do, and freely agree to take part in the interview, please give consent to be interviewed and have your responses used.

UNIVERSITY OF ALABAMA IRB
CONSENT FORM APPROVED: 4-23-15
EXPIRATION DATE: 4-22-16

August 19, 2010

Office for Research

Institutional Review Board for the
Protection of Human Subjects

Jeffrey C. Carver, Ph.D.
Department of Computer Science
College of Engineering
The University of Alabama

Re: IRB # 10-OR-263 "Survey of Software Engineering Practices in Computational Science"

Dear Dr. Carver:

The University of Alabama Institutional Review Board has granted approval for your proposed research

Your application has been given expedited approval according to 45 CFR part 46. You have also been granted the requested waiver of written documentation of informed consent. Approval has been given under expedited review category 7 as outlined below:

(7) Research on individual or group characteristics or behavior (including, but not limited to, research on perception, cognition, motivation, identity, language, communication, cultural beliefs or practices, and social behavior) or research employing survey, interview, oral history, focus group, program evaluation, human factors evaluation, or quality assurance methodologies.

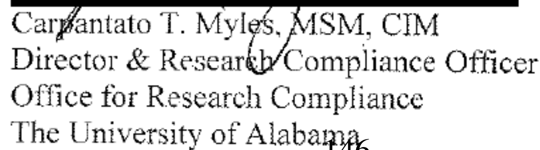
Your application will expire on August 16, 2011. If your research will continue beyond this date, complete the relevant portions of Continuing Review and Closure From. If you wish to modify the application, complete the Modification of an Approved Protocol Form. When the study closes, complete the appropriate portions of FORM: Continuing Review and Closure.

Please use reproductions of the IRB approved informed consent form to obtain consent from your participants.

Should you need to submit any further correspondence regarding this proposal, please include the above application number.

Good luck with your research.

Sincerely,



Carpantato T. Myles, MSM, CIM
Director & Research Compliance Officer
Office for Research Compliance
The University of Alabama

152 Rose Administration Building
Box 870117
Tuscaloosa, Alabama 35487-0117
(205) 348-8461
FAX (205) 348-8882
TOLL FREE (877) 820-3066

IRB Project #: 10-OR-263

UNIVERSITY OF ALABAMA
INSTITUTIONAL REVIEW BOARD FOR THE PROTECTION OF HUMAN SUBJECTS
REQUEST FOR APPROVAL OF RESEARCH INVOLVING HUMAN SUBJECTS

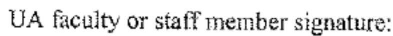
I. Identifying information

	Principal Investigator	Second Investigator	Third Investigator
Names:	Jeffrey Carver		
Department:	Computer Science		
College:	Engineering		
University:	Alabama		
Address:	Box 870290		
Telephone:	8-9829		
FAX:	8-0219		
E-mail:	carver@cs.ua.edu		

Title of Research Project: Survey of Software Engineering Practices in Computational Science

Date Submitted: July 22, 2010
Funding Source: N/A

Type of Proposal	☒ New	☐ Revision	☐ Renewal Please attach a renewal application	☐ Completed	☐ Exempt
			Please attach a continuing review of studies form		
		Please enter the original IRB # at the top of the page			

UA faculty or staff member signature: 

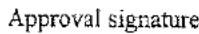
II. NOTIFICATION OF IRB ACTION (to be completed by IRB):
Type of Review: ______ Full board __✓ Expedited

IRB Action:

___ Rejected Date: ________
___ Tabled Pending Revisions Date: ________
___ Approved Pending Revisions Date: ________
✓ Approved-this proposal complies with University and federal regulations for the protection of human subjects.

Approval is effective until the following date: 8-16-11/ss

Items approved: ___ Research protocol (dated__________)
___ Informed consent (dated__________)
___ Recruitment materials (dated__________)
[...]d__________)

Approval signature  Date 8/17/2010

UNIVERSITY OF ALABAMA
Informed Consent for a Research Study

You are being asked to take part in a research study. This study is called *Survey of Software Engineering Practices in Computational Science*. The study is being done by *Dr. Roscoe Barnett, Sandia Laboratory, Dr. Jeffrey Carver,* University of Alabama, and *Dr. Lorin Hochstein, University of Southern California*.

What is this study about?
This survey will help us understand your current practices related to software development in computational science.

Why is this study important--What good will the results do?
This study will help us better what software development practices computational scientists are currently using and where they have needs for more information.

Why have I been asked to take part in this study?
You have been asked to be in this study because you are subscribed to a relevant mailing list.

How many people besides me will be in this study?
Approximately 100 people will participate in the study.

What will I be asked to do in this study?
If you decide to be in this study, you will be asked to complete a survey.

How much time will I spend being in this study?
Approximately 15 minutes.

Will I be paid for being in this study?
No.

Will being in this study cost me anything?
There will be no cost to you.

Can the researcher take me out of this study?
The researcher may take you out of the study if your survey is incomplete.

What are the benefits (good things) that may happen to me if I am in this study?
There are no direct benefits to you from being in this study.

What are the benefits to scientists or society?
This study will help the researchers understand current computational science software development practices and areas of future need.

What are the risks (dangers or harm) to me if I am in this study?
There are no risks to participating in this study. All information will be anonymous.

How will my confidentiality (privacy) be protected? What will happen to the information the study keeps on me?
The surveys will be anonymous.

What are the alternatives to being in this study? Do I have other choices?
The alternative/other choice is not to participate.

What are my rights as a participant?
Taking part in this study is voluntary. You may choose not to take part at all. If you start the study, you can stop at any time. Leaving the study will not result in any penalty or loss of any benefits you would otherwise receive.

Who do I call if I have questions or problems?
If you have questions about the study, please contact Dr. Carver at (205)-348-9828 or carver@cs.ua.edu. If you have any questions about your rights as a research participant you may contact Ms. Tanta Myles, The University of Alabama Research Compliance Officer, at (205)-348-8461 or 1-877-820-3066.

UA IRB Approved Document
Approval date: 8-17-10
Expiration date: 8-16-11

If you have read this consent form, had the study adequately explained to you, understand what you are being asked to do, and freely agree to take part in the survey, click the "next" button. Otherwise, please exit the survey.

UA IRB Approved Document
Approval date: 8-17-10
Expiration date: 8-16-11