Information and Software Technology 67 (2015) 207–219

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Claims about the use of software engineering practices in science: A systematic literature review

Dustin Heaton, Jeffrey C. Carver[*]

Department of Computer Science, University of Alabama, Tuscaloosa, Alabama, USA

ARTICLE INFO

Article history:
Received 26 March 2015
Revised 15 June 2015
Accepted 27 July 2015
Available online 7 August 2015

Keywords:
Computational science
Systematic literature review
Scientific software

ABSTRACT

Context: Scientists have become increasingly reliant on software in order to perform research that is too time-intensive, expensive, or dangerous to perform physically. Because the results produced by the software drive important decisions, the software must be correct and developed efficiently. Various software engineering practices have been shown to increase correctness and efficiency in the development of traditional software. It is unclear whether these observations will hold in a scientific context.

Objective: This paper evaluates claims from software engineers and scientific software developers about 12 different software engineering practices and their use in developing scientific software.

Method: We performed a systematic literature review examining claims about how scientists develop software. Of the 189 papers originally identified, 43 are included in the literature review. These 43 papers contain 33 different claims about 12 software engineering practices.

Results: The majority of the claims indicated that software engineering practices are useful for scientific software development. Every claim was supported by evidence (i.e. personal experience, interview/survey, or case study) with slightly over half supported by multiple forms of evidence. For those claims supported by only one type of evidence, interviews/surveys were the most common. The claims that received the most support were: "The effectiveness of the testing practices currently used by scientific software developers is limited" and "Version control software is necessary for research groups with more than one developer." Additionally, many scientific software developers have unconsciously adopted an agile-like development methodology.

Conclusion: Use of software engineering practices could increase the correctness of scientific software and the efficiency of its development. While there is still potential for increased use of these practices, scientific software developers have begun to embrace software engineering practices to improve their software. Additionally, software engineering practices still need to be tailored to better fit the needs of scientific software development.

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

[*] Corresponding author. Tel.: +1- 205- 348- 9829.
E-mail addresses: dwheaton@crimson.ua.edu (D. Heaton), carver@cs.ua.edu (J.C. Carver).

http://dx.doi.org/10.1016/j.infsof.2015.07.011
0950-5849/© 2015 Elsevier B.V. All rights reserved.

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 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 [1]. The size of the programs also increases the complexity, as scientific software can contain more than 100,000 lines of code [2,3]. 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. Section 2 provides more details about these characteristics of scientific software.

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 manage 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).

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 [4].

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 interaction 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 (Section 2) 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 analysis of the effectiveness of those practices; and
3. An analysis of the evidence used to show effectiveness.

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 in the software engineering literature. 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 provides background on previous research about SE for scientific software. Section 3 describes the research methodology used in this systematic literature review. Section 4 reports the scientists' and software engineers' claims about SE for scientific software.

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 [1]. Furthermore, even for developers who place a great deal of emphasis on software quality, it is likely that at least some existing software engineering practices must be tailored to be effective in scientific software development [5].

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. [6] enumerated three characteristics that are common across the majority of the community. In addition to these common characteristics, Basili et al. [6] also enumerate three variables that differentiate projects within the scientific software community. The following subsections describe the common and variable aspects, respectively.

2.1. Common characteristics of scientific software development

According to Basili et al., [6], there are three characteristics that 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 [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