# 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*
*dwheaton@crimson.ua.edu, carver@cs.ua.edu*

**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.

*Keywords:* Computational Science, Systematic Literature Review, Scientific Software

---

## 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 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 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).

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 primarly 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 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 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 [8] also observed this characteristic.

3. **Typical user base** - Most scientific software (with the exception of some libraries and large commercially-avaiable software pacages - 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. 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., [6] 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 users, 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 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).

## 3. Methodology

The following subsections describe the steps of the Systematic Literature Review (SLR) process we followed [9].

### *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?*

### *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 string, *"scientific software development" AND "software engineering"* resulted in a more manageable 349 papers. After reviewing these papers, we identified the list of applicable software engineering topics described in Table 4. To help ensure the completeness of our search, we repeated the search by replacing "software engineering" with each of these 11 terms. These additional search strings resulted in the identification of a total of 718 papers. Some of which were duplicates of those identified in the initial search. We conducted this search throughout May 2015.

### *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 1.

Table 1: Inclusion and exclusion criteria

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

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. Table 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.

### *3.4. Data Extraction*

Table 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 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 [10, 11, 12, 13, 14].

Table 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 Discreet Mathematics | 1 |
| IEEE International Conference on Electro/Information Technology | 1 |

Table 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 |

## 4. Results

In our review of the literature, we used the definitions provided by the IEEE Standard Computer Dictionary [15] 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 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 seperate answers for each research question, in this review we believed it made more sense to answer them together so that each claim would be 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.

### *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 7-11. First the claim is presented and then the papers that made the claim are

Table 4: SE practices

| | |
|---|---|
| **Development Workflow** | Design Issues<br>Lifecycle Model<br>Documenatation<br>Refactoring<br>Requirements<br>Testing<br>Verification and Validation |
| **Infrastructure** | Issue Tracking<br>Reuse<br>Third-Party Issues<br>Version Control |

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.

#### *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 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 [16, 2, 3, 17, 18, 19, 20, 21]. 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." [16]

**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 [2, 3, 20, 22, 21, 23, 24]. When scientific 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 [2]. Scientific software developers support the observation that they generally use an agile development approach because they do not know the requirements ahead of time [3].

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 [17].

**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 code [17]. Conversely, some scientists also say that paired programming is not natural for scientific software developers, and so it is not useful in all cases [25, 18].

**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 [18],
2. **Test-driven development** reduces the number of errors introduced into the code for multiple teams [19, 21, 26], 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 [16, 27]. 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 [28].

**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 [5]. 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 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 [5].

#### *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 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 [29, 18, 5, 16, 30, 31]. 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 [30]. In another example, scientists developed using an iterative approach which allowed the

Table 5: Lifecycle Model

| Claim | NS | PE | I/S | CS |
|---|---|---|---|---|
| LM1: Scientific software developers do not generally use a formal software development methodology | [19] | [16, 17, 24] | | [2, 3, 18, 20] |
| LM2: The development methodology used by scientific software developers is similar to the agile development methodology | | [17, 22, 23] | [24] | [2, 3, 20, 27, 21] |
| LM3: Paired programming provides an effective practice for dealing with complex software development requirements and an effective avenue for knowledge transfer | | [17, 25] | | [18] |
| LM4: Scientific software developers viewed feature-driven development, test-driven development, and iterative development as effective | [19, 28] | [16] | | [18, 21, 26] |
| LM5: Existing software development methodologies will need to be tailored to the specific context of any given scientific software development team | | [5] | | |

Table 6: Requirements

| Claim | NS | PE | I/S | CS |
|---|---|---|---|---|
| RQ1: Scientific software developers often do not produce proper requirements specifications | | [16] | [29, 30] | [18, 5, 31] |
| RQ2: When scientific software developers produce requirements, they generally focus on high-level requirements | | [16] | | [31] |
| RQ3: When scientists produce high-level requirements they rely on developers to prioritize them | | | [29] | |

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 [5].

**RQ2: When scientific software developers to produce requirements, they generally focus on high-level requirements.** We found two studies that made this claim [16, 31] 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 [16].

**RQ3: When scientists produce high-level requirements, they rely on developers to prioritize them.** In one case we found [29], the lack of scientific background made it difficult for the software developers to properly prioritize requirements and effectively develop the software [29]. To rectify this problem, a scientific representative had to be assigned in a later stage to prioritize requirements for the developers [29].

*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 [29]. We group the detailed list of claims into three over-arching claims in the following

discussion. Table 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 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 [29].

**DI3:Object Oriented Design helps produce useful software.** The other three studies [32, 33, 34] 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.

Table 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 | | | [29] | |
| DI2: Scientific software developers see re-design as a waste of time that risks breaking the "science" of a program | | | [29] | |
| DI3: Object Oriented Design helps produce useful software | | [32, 33, 34] | | |

#### *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 claims in the following discussion. Table 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 [19, 35, 36, 37, 3, 16, 38, 39, 40, 41]. One of these studies was a survey, the results of which are summarized in Figure 1.

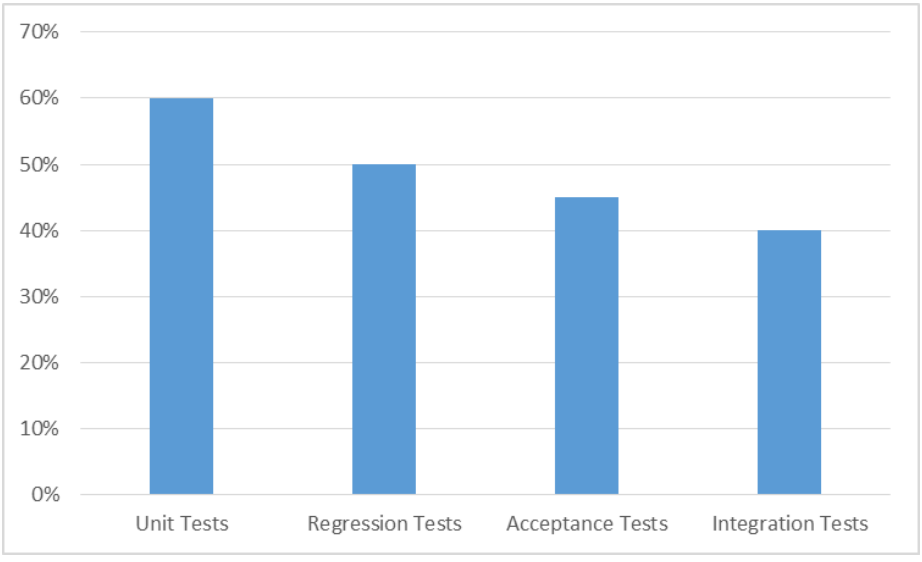


Figure 1: Testing Types from Survey

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'
- 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 [38].

**T2: Scientific software developers benefit from using a wide range of testing practices from software engineering.** Twelve studies made this claim [19, 28, 37, 42, 40, 39, 41, 43, 44, 45, 46, 47]. 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 [19, 37, 39, 42]. 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 [28, 37]. 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 [3, 40].

**T3: The testing practices that scientific software developers do utilize are often executed 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 [45, 47]. 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 [30].

**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 [3, 16, 48, 23]. 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 [19, 37]. 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 [48]

Table 8: Testing

| Claim | NS | PE | I/S | CS |
|---|---|---|---|---|
| T1: The effectiveness of the testing practices currently used by scientific software developers is limited | [19] | [37, 40] | [38] | [35, 36, 3, 16, 39, 41] |
| T2: Scientific software developers would benefit from using a wide range of testing practices from software engineering | [19, 28] | [37, 42, 40, 43, 47] | | [46, 39, 41, 44, 45] |
| T3: The testing practices that scientific software developers do utilize are often executed poorly. | | [45, 47] | [30] | |
| 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 | [19] | [16, 37, 48, 23] | | [3] |

*4.1.5. Verification and Validation*

Our survey of the literature identified eleven studies that contained claims about the use of Verification and Validation (V&V) by scientific software developers. It is worth nothing that software engineers and scientific software developers may have slightly different understandings of V&V. Based on the definitions provided by Babuska and Oden [49], we provide a mapping between the two domains.

First, software engineers typically understand *Validation* as the process of ensuring that the project specification (and overall end product) matches the project goals or user requirements. In scientific software, this same concept is described as ensuring that the mathematical model (the equivalent of the project specification) matches the real world (the equivalent of the project goals or user requirements).

Second, software engineers typically understand *Verification* as the process of ensuring that the implementation of the software matches the project specification. In scientific software, this same concept is described as ensuring that the computational model (the equivalent of the software implementation) matches the mathematical model (the equivalent of the project spectification).

We group the detailed list of claims into four over-arching claims in the following discussion. Table 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 [3, 50, 39, 23, 46]. 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 [51, 50, 39, 23, 46]. 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 [51]. 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 [51].

**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, 52, 39].

**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 [53].

**VV4: Experimental validation is frequently impractical because scientists lack the information they would prefer to use to validate the software.** We found this claim in four studies [3, 2, 54, 55]. 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 [3]. 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 [2, 54, 55]. In some cases, scientific software developwers 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 [55].

Table 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 | | [50, 39, 23, 46] | | [3, 51] |
| VV2: There are many ways that defects can enter software | | [39] | | [7, 52] |
| VV3: Scientists frequently suspect that any problems in the results of their software result from scientific theory | | [53] | | |
| VV4: Experimental validation is frequently impractical because scientists lack the information they would prefer to use to validate the software | | | [54, 55] | [3, 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-arching claims in the following discussion. Table 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 [56, 57],
- has proven to be a highly valuable practice for the bioinformatics domain [56],
- is also a powerful practice for maintaining and improving the quality of code [56, 58], and
- is particularly useful in conjunction with the automated refactoring tools of IDEs such as Eclipse [19].

Table 10: Refactoring

| Claim | NS | PE | I/S | CS |
|---|---|---|---|---|
| RF1: Refactoring is a useful practice to increase software quality | [19] | [57] | | [56, 58] |
| RF2: Refactoring is not always possible | | | | [3] |

**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 [3].

*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 over-arching claims in the following discussion. Table 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 [25, 59, 8, 23]. 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 [28, 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 [35, 58, 8].

**D2: Documentation is becoming more frequently used.** In a more recent study, Nguyen-Hoan et al. [38] 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. 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' [38]."

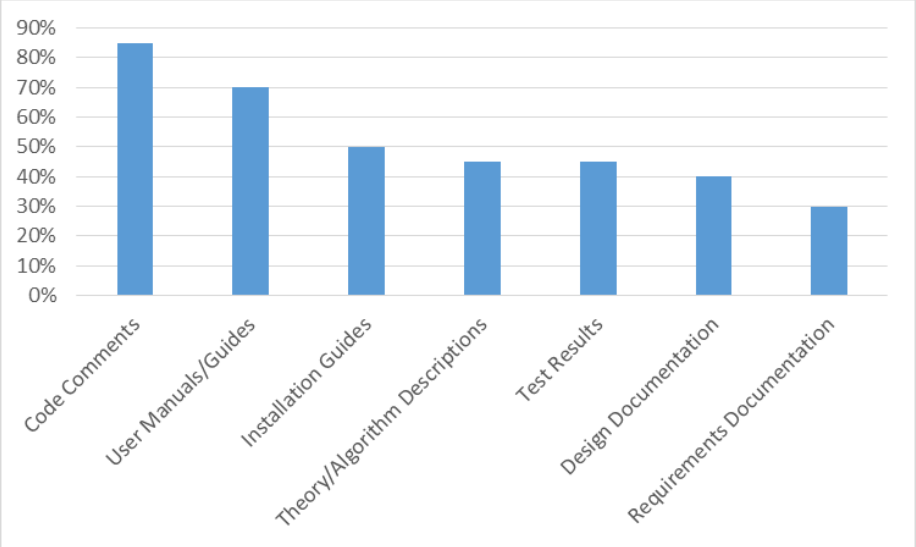


Figure 2: Documentation Produced by Developers

**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 documentation they create [25, 59]. Furthermore, if the documentation is done to satisfy an external requirement it may not benefit the project team [25, 59]. 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

Table 11: Documentation

| Claim | NS | PE | I/S | CS |
|---|---|---|---|---|
| D1: Documentation is a necessary enabler of software quality | [28] | [25, 8, 23] | [59] | [52, 35, 58] |
| D2: Documentation is becoming more frequently used | | | [38] | |
| D3: Documentation requires a significant investment of work | | [25] | [59] | [35] |

the project's source files [35].

*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 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.

### *4.2. Infrastructure*

The practices in this category all serve to support various aspects of the software development lifecycle. Each of practice is addressed in its own section and the claims are summarized in Tables 12-15. The tables use the same notation as described in Section 4.1

#### *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 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 [19]. Issue-tracking software also made a list of the ten most important software engineering practices for scientific software developers to use [25]. 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 [25].

Table 12: Issue Tracking

| Claim | NS | PE | I/S | CS |
|---|---|---|---|---|
| IT1: Issue Tracking greatly eases communication between members of a development team. | [19] | [25] | | |
| IT2: Issue Tracking helps insure that no two groups in a development team are working on the same problem | | [25] | | |

**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 can then be distributed among various groups so that no two groups are working on the same feature [25].

*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 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 [8]. The most reused types of artifacts are source code, scripts, algorithms, and practices [60, 20, 33, 61, 62, 63].

**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 many reasons for

Table 13: Reuse

| Claim | NS | PE | I/S | CS |
|---|---|---|---|---|
| RU1: Software must be properly designed to be reusable | [20] | [60, 33, 63, 8] | [61] | [62] |
| RU2: There are many reasons not to reuse or produce reusable software | | [60, 64] | [61] | [62] |

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 [60, 64, 61, 62].

*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 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 [6, 65]. 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 [6, 65]. Additionally, a significant barrier to the use of existing frameworks is that they cannot be integrated 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 [6].

**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 [2, 20, 66, 22, 67, 65, 62, 63]. 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 [2].

**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 [66].

#### *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 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 [3, 19, 35, 4, 6, 53, 55, 27]. Version control tools 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

Table 14: Third-Party Issues

| Claim | NS | PE | I/S | CS |
|---|---|---|---|---|
| TPI1: Third Party Software may cease being supported before scientific software projects are finished | | [65] | [6] | |
| TPI2: Scientific software developers are not convinced that reusing existing frameworks will save effort in their development | | [65] | [6] | |
| TPI3: Open-source software is especially useful to scientific software developers | | [66, 22, 63] | | [2, 20, 67, 65, 62] |
| TPI4: Open-sourcing can be seen as giving up competitive advantage | | [66] | | |

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 [53]. 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 [53]. 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 [55].

**VC2: Distributed Version Control is particularly useful for scientific software development.** 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

Table 15: Version Control

| Claim | NS | PE | I/S | CS |
|---|---|---|---|---|
| VC1: Version control software is necessary for research groups with more than one developer | [19] | [4, 53] | [6, 55] | [3, 35, 27] |
| VC2: Distributed Version Control is particularly useful for scientific software development | [19] | | | [35, 3] |

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 [68]. 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 [35]. 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 [3, 19].

#### *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.

## 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 developers. 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 need adopting software engineering practices would allow them to produce higher quality 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 closly 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 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.

**Acknowledgments**

The authors acknowledge support from NSF grant 1445344.

**References**

[1] J. C. Carver, Se-CSE 2008: The first international workshop on software engineering for computational science and engineering, in: Companion of the 30th International Conference on Software Engineering, ICSE Companion '08, ACM, New York, NY, USA, 2008, pp. 1071–1072. doi:10.1145/1370175.1370252.

[2] J. C. Carver, R. P. Kendall, S. E. Squires, D. E. Post, Software development environments for scientific and engineering software: A series of case studies, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 550–559. doi:10.1109/ICSE.2007.77.

[3] S. M. Easterbrook, T. C. Johns, Engineering the software for understanding climate change, Computing in Science and Engineering 11 (2009) 65–74.

[4] S. Faulk, E. Loh, M. L. Vanter, S. Squires, L. G. Votta, Scientific computing's productivity gridlock: How software engineering can help, Computing in Science and Engineering 11 (2009) 30–39.

[5] J. Segal, When software engineers met research scientists: A case study, Empirical Software Engineering 10 (2005) 517–536.

[6] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, M. Zelkowitz, Understanding the high-performance-computing community: A software engineer's perspective, Software, IEEE 25 (2008) 29–36.

[7] J. C. Carver, Report from the second international workshop on software engineering for computational science and engineering, Computing in Science and Engineering 11 (2009) 14–19.

[8] K. Hinsen, Software development for reproducible research, Computing in Science and Engineering 15 (2013) 60–63.

[9] B. Kitchenham, S. Charters, Guidelines for performing Systematic Literature Reviews in Software Engineering, Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007. URL: `http://www.dur.ac.uk/ebse/resources/Systematic-reviews-5-8.pdf`.

[10] B. J. Williams, J. C. Carver, Characterizing software architecture changes: A systematic review, Information and Software Technology 52 (2010) 31–51.

[11] J. E. Hannay, D. I. K. Sjoberg, T. Dyba, A systematic review of theory use in software engineering experiments, IEEE Trans. Softw. Eng. 33 (2007) 87–107.

[12] M. Jorgensen, M. Shepperd, A systematic review of software development cost estimation studies, IEEE Trans. Softw. Eng. 33 (2007) 33–53.

[13] B. Kitchenham, E. Mendes, G. H. Travassos, Cross versus within-company cost estimation studies: A systematic review, Software Engineering, IEEE Transactions on 33 (2007) 316–329.

[14] E. Insfran, A. Fernandez, A systematic review of usability evaluation in web development, in: S. Hartmann, X. Zhou, M. Kirchberg (Eds.), Web Information Systems Engineering WISE 2008 Workshops, volume 5176 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2008, pp. 81–91. doi:`10.1007/978-3-540-85200-1_10`.

[15] IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries, IEEE Std 610 (1991) 1–217.

[16] J. Segal, Some challenges facing software engineers developing software for scientists, ICSE Workshop on Software Engineering for Computational Science and Engineering, 2009.

[17] S. Killcoyne, J. Boyle, Managing chaos: Lessons learned developing software in the life sciences, Computing in Science and Engineering 11 (2009) 20–29.

[18] R. Kendall, J. Carver, D. Fisher, D. Henderson, A. Mark, D. Post, Development of a weather forcasting code: A case study, IEEE Software 25 (2008) 59–65.

[19] K. Ackroyd, S. Kinder, G. Mant, M. Miller, C. Ramsdale, P. Stephenson, Scientific software development at a research facility, IEEE Software 25 (2008) 44–51.

[20] J. Y. Monteith, J. D. McGregor, J. E. Ingram, Scientific research software ecosystems, in: Proceedings of the 2014 European Conference on Software Architecture Workshops, ECSAW '14, ACM, New York, NY, USA, 2014, pp. 9:1–9:6. doi:10.1145/2642803.2642812.

[21] A. Nanthaamornphong, K. Morris, D. Rouson, H. Michelsen, A case study: Agile development in the community laser-induced incandescence modeling environment (cliime), in: Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on, 2013, pp. 9–18. doi:10.1109/SECSE.2013.6615094.

[22] S. Ahalt, B. Minsker, M. Tiemann, L. Band, M. Palmer, R. Idaszak, C. Lenhardt, 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, SE-CSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 40–47. URL: http://dl.acm.org/citation.cfm?id=2663370.2663377.

[23] D. Kelly, S. Smith, N. Meng, Software engineering for scientists, Computing in Science and Engineering 13 (2011) 7–11.

[24] M. T. Sletholt, J. Hannay, D. Pfahl, H. C. Benestad, 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, SECSE '11, ACM, New York, NY, USA, 2011, pp. 1–9. doi:10.1145/1985782.1985784.

[25] M. A. Heroux, 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, SECSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 15–21.

[26] M. Rilee, T. Clune, Towards test driven development for computational science with pfunit, in: Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), 2014 Second International Workshop on, 2014, pp. 20–27. doi:10.1109/SE-HPCCSE.2014.5.

[27] R. Betz, R. Walker, Streamlining development of a multimillion-line computational chemistry code, Computing in Science and Engineering 16 (2014) 10–17.

[28] D. Post, R. Kendal, 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 18 (2004) 399–416.

[29] 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 18 (2009) 581–606.

[30] R. Sanders, D. Kelly, Dealing with risk in scientific software development, IEEE Software 25 (2008) 21–28.

[31] Y. Li, N. Narayan, J. Helming, M. Koegel, A domain specific requirements model for scientific computing (nier track), in: Proceedings of the 33rd

International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 848–851. doi:10.1145/1985793.1985922.

[32] A. George, 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 20 (1999) 953–969.

[33] L. Hall, C. Hung, C. Hwang, A. Oyake, J. Yin, COTS-based oo-component approach for software inter-operability and reuse (software systems engineering methodology), in: Aerospace Conference, 2001, IEEE Proceedings., volume 6, 2001, pp. 2871–2878vol.6. doi:10.1109/AERO.2001.931308.

[34] F. Pluquet, S. Langerman, A. Marot, R. Wuyts, Implementing partial persistence in object-oriented languages, in: 2008 Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX), 2008, pp. 37–48. doi:10.1137/1.9781611972887.4.

[35] Y. Pouillon, J. Beuken, T. Deutsch, M. Torrent, X. Gonze, Organizing software growth and distributed development: The case of abinit, Computing in Science and Engineering 12 (2011) 62–69.

[36] C. Morris, J. Segal, Some Challenges Facing Scientific Software Developers: The Case of Molecular Biology, Fifth IEEE International Conference on e-Science, 2009.

[37] P. Dubois, Maintaining correctness in scientific programs, Computing in Science and Engineering 7 (2005) 80–85.

[38] L. Nguyen-Hoan, S. Flint, R. Sankaranarayana, A survey of scientific software development, in: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '10, ACM, New York, NY, USA, 2010, pp. 12:1–12:10.

[39] H. Remmel, B. Paech, C. Engwer, P. Bastian, Design and rationale of a quality assurance process for a scientific framework, in: Software Engineer-

ing for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on, 2013, pp. 58–67. doi:10.1109/SECSE.2013.6615100.

[40] M. Miic, M. Tomaevic, I. Bethune, Automated multi-platform testing and code coverage analysis of the CP2K application, in: Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on, 2014, pp. 95–98. doi:10.1109/ICST.2014.21.

[41] K. Karhu, T. Repo, O. Taipale, K. Smolander, Empirical observations on software testing automation, in: Proceedings of the 2009 International Conference on Software Testing Verification and Validation, ICST '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 201–209. doi:10.1109/ICST.2009.16.

[42] C. Omar, J. Aldrich, R. C. Gerkin, Collaborative infrastructure for test-driven scientific model validation, in: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, ACM, New York, NY, USA, 2014, pp. 524–527. doi:10.1145/2591062.2591129.

[43] C. Martínez, L. Moura, D. Panario, B. Stevens, Locating errors using elas, covering arrays, and adaptive testing algorithms, SIAM Journal on Discrete Mathematics 23 (2010) 1776–1799.

[44] R. Nori, N. Karodiya, H. Reza, Portability testing of scientific computing software systems, in: Electro/Information Technology (EIT), 2013 IEEE International Conference on, 2013, pp. 1–8. doi:10.1109/EIT.2013.6632686.

[45] D. Kelly, S. Thorsteinson, D. Hook, Scientific software testing: Analysis with four dimensions, Software, IEEE 28 (2011) 84–90.

[46] H. Remmel, B. Paech, P. Bastian, C. Engwer, System testing a scientific framework using a regression-test environment, Computing in Science and Engineering 14 (2012) 38–45.

[47] P. Dubois, Testing scientific programs, Computing in Science and Engineering 14 (2012) 69–73.

[48] U. Kanewala, J. Bieman, Techniques for testing scientific programs without an oracle, in: Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on, 2013, pp. 48–57. doi:10.1109/SECSE.2013.6615099.

[49] I. Babuska, J. T. Oden, Verification and validation in computational engineering and science: Basic concepts, Computer Methods in Applied Mechanics and Engineering 193 (2004) 4057–4066.

[50] D. Higdon, M. Kennedy, J. C. Cavendish, J. A. Cafeo, R. D. Ryne, Combining field data and computer simulations for calibration and prediction, SIAM Journal on Scientific Computing 26 (2004) 448–466.

[51] D. Post, R. Kendall, R. Lucas, The Opportunities, Challenges and Risks of High Performance Computing in Computational Science and Engineering, Advances in Computers, 2006.

[52] J. C. Carver, L. M. Hochstein, R. P. Kendall, T. Nakamura, M. V. Zelkowitz, V. R. Basili, D. E. Post, Observations about software development for high end computing, CTWatch Quarterly 2 (2006) 33–37.

[53] D. Kelly, D. Hook, R. Sanders, Five recommended practices for computational scientists who write software, Computing in Science and Engineering 11 (2009) 48–53.

[54] F. Shull, J. Carver, L. Hochstein, V. Basili, Emperical study design in the area of high-performance computing (HPC), International Symposium on Emperical Software Engineering, 2005.

[55] L. Hochstein, V. R. Basili, The asc-alliance projects: A case study of large-scale parallel scientific code development, Computer 41 (2008) 50–58.

[56] C. Crabtree, A. Koru, C. Seaman, H. Erdogmus, An empirical characterization of scientific software development projects according to the boehm and turner model: A progress report, in: Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on, 2009, pp. 22–27. doi:10.1109/SECSE.2009.5069158.

[57] W. R. Elwasif, B. R. Norris, B. A. Allan, 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, CompFrame '07, ACM, New York, NY, USA, 2007, pp. 21–30. doi:10.1145/1297385.1297390.

[58] Y. Li, Reengineering a scientific software and lessons learned, in: Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering, SECSE '11, ACM, New York, NY, USA, 2011, pp. 41–45. URL: http://doi.acm.org/10.1145/1985782.1985789. doi:10.1145/1985782.1985789.

[59] A. Pawlik, J. Segal, M. Petre, Documentation practices in scientific software development, in: Cooperative and Human Aspects of Software Engineering (CHASE), 2012 5$^{th}$ International Workshop on, 2012, pp. 113–119. doi:10.1109/CHASE.2012.6223004.

[60] S. Samadi, N. Almaeh, R. Wolfe, S. Olding, D. Issac, Strategies for enabling software reuse within the earth science community, IEEE International Geoscience and Remote Sensing Symposium 25 (2008) 2196–2199.

[61] E. H. Trainer, C. Chaihirunkarn, A. Kalyanasundaram, J. D. Herbsleb, From personal tool to community resource: What's the extra work and who will do it?, in: Proceedings of the 18$^{th}$ ACM Conference on Computer Supported Cooperative Work & Social Computing, CSCW '15, ACM, New York, NY, USA, 2015, pp. 417–430. doi:10.1145/2675133.2675172.

[62] X. Huang, X. Ding, C. P. Lee, T. Lu, N. Gu, Meanings and boundaries of scientific software sharing, in: Proceedings of the 2013 Conference on

Computer Supported Cooperative Work, CSCW '13, ACM, New York, NY, USA, 2013, pp. 423–434. doi:10.1145/2441776.2441825.

[63] D. I. Ketcheson, K. Mandli, A. J. Ahmadia, A. Alghamdi, M. Q. de Luna, M. Parsani, M. G. Knepley, M. Emmett, Pyclaw: Accessible, extensible, scalable tools for wave propagation problems, SIAM Journal on Scientific Computing 34 (2012) C210–C231.

[64] E. H. Trainer, C. Chaihirunkarn, A. Kalyanasundaram, 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, GROUP '14, ACM, New York, NY, USA, 2014, pp. 111–121. doi:10.1145/2660398.2660420.

[65] D. Matthews, G. Wilson, S. Easterbrook, Configuration management for large-scale scientific computing at the uk met office, Computing in Science and Engineering 10 (2008) 56–64.

[66] 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, XSEDE '13, ACM, New York, NY, USA, 2013, pp. 69:1–69:7. doi:10.1145/2484762.2484782.

[67] I. Gorton, Y. Liu, C. Lansing, T. Elsethagen, 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, CBSE '13, ACM, New York, NY, USA, 2013, pp. 159–168. doi:10.1145/2465449.2465460.

[68] K. Hinsen, K. Laufer, G. K. Thiruvathukal, Essential tools: Version control systems, Computing in Science and Engineering 11 (2009) 84–91.