# What Scientists and Engineers Think They Know About Software Engineering: A Survey

Jeffrey Carver[1], Roscoe Bartlett[2], Dustin Heaton[1] and Lorin Hochstein[3]

[1]University of Alabama
carver@cs.ua.edu;
dwheaton@ua.edu;

[2]Sandia National Laboratories
rbartl@sandia.gov

[3]USC-ISI
lorin@isi.edu

*Scientists and engineers devote considerable effort to developing large, complex codes to solve important problems. Our personal experience with such teams suggested that, while they often develop good code, many of these developers are frequently unaware of how various software engineering practices can help them write better code. Our hypothesis is that many of these developers "don't know what they don't know," as was the case for one of the authors of this article. To test this hypothesis, we conducted a survey of computational scientists and engineers. We received 141 responses to the survey. The first main finding of the survey was that most developers were largely self-taught. The second main finding was that while most respondents thought they knew enough software engineering to be effective, many were not familiar with standard software engineering practices used in commercial industry*.

## Introduction

As readers of this magazine know, scientists and engineers must often double as part-time software developers, performing tasks that range from relatively simple and small-scale (i.e. dashing off a quick Python script for parsing a data file) all the way to very large and complex (i.e. maintaining fusion reaction simulation codes that are hundreds of thousands of lines, integrating libraries written in Fortran, C, and C++, and running on the latest supercomputer). The quality of scientific software can vary enormously, as was vividly illustrated after a server at the Climate Research Unit of the University of East Anglia was hacked in November 2009. The released documents included IDL scripts to generate plots from weather data and an accompanying readme file which contained the line "Something is very poorly [sic]. It's my programming skills, isn't it" [1]. On A BBC *Newsnight* broadcast on December 3, a professional software developer who examined the IDL code in detail judged its quality as being below that of commercial source code [2] and subsequently identified calculation errors in the dataset that led to corrections by the UK Met Office [3]. These discrepancies occurred despite positive findings from recent case studies of climate model codes which revealed the use of very sophisticated verification and validation methodologies [4].

Even though software development skills are becoming as integral to science and engineering as laboratory skills, the community has not evolved to have a culture of training for the relevant software skills. All too often, scientists and engineers possess just enough programming knowledge to get by, but are not exposed to software development as practiced in the commercial world. As a result, *they don't know what they don't know*. As a specific example of this knowledge problem, one of the authors of this paper, Ross Bartlett, who comes from the Chemical Engineering and Applied Math wing of the CSE (Computational Science and

Engineering) community had the personal experience of "not knowing what he did not know" as it related to the use of software engineering on his projects. He also observed that many of his CSE colleagues had the same problem. When Bartlett begun his work at Sandia National Labs in 2001, he thought that he knew enough software engineering to produce software of reasonable quality. He thought that he was, in fact, producing quality software. It was not until years later (starting in 2007) that he began reading a larger portion of the modern software engineering literature (see http://www.cs.sandia.gov/~rabartl/readingList.html) and came to realize his own knowledge gap in best practices for creating and maintaining quality software. After this realization, he began applying many of the software engineering best practices to his CSE projects and realized significant improvements in his productivity and the overall quality of his software. He was also horrified to realize that, prior to gaining this knowledge, he had created tens of thousands of lines of CSE code of insufficient quality that he was now in a position to have to help maintain. As he examined many other CSE projects in which he was involved, he saw that they were in the same place he was years before, i.e. incorrectly believing that they knew how to produce quality software and were actually producing quality software. He began to wonder whether his personal experience in regards to software engineering knowledge was common in the broader CSE community.

Based on these observations, Bartlett was interested in gaining a broader view of the CSE community's understanding of traditional software engineering principles. The best way to gather information from a broad community is via survey. Bartlett contacted co-authors Carver and Hochstein to help in developing this survey. Carver and Hochstein come from the software engineering community and have been researching the use of software engineering principles in the development of CSE software. Carver and Hochstein also have experience in capturing the software engineering practices of CSE software teams [5-8]. Taking advantage of our complementary expertise, this collaboration enabled us to develop an appropriate survey. The survey sought to provide us with an understanding of the level of knowledge and use of various software engineering practices by members of the CSE community. The goal of this survey is to get a general understanding of the current state of the CSE community with an eye towards developing more specific and detailed surveys for future distribution.

In the remainder of this article, we first describe the survey, including the questions it contained and the recipients. Then we describe observations we made from the data obtained. Finally, we draw conclusions about the results and discuss future directions of this work.

## The Survey

The goal of the survey was to gather the respondents' perception of software engineering knowledge and use at three different levels. First, we wanted to understand how the survey respondents assessed *themselves* in terms of their software engineering skills. Second, we were curious about how the survey respondents assessed their *teammates'* level of software engineering knowledge and skill. Finally, we were interested in the survey respondents' opinions of the level of software engineering knowledge and skill for the *CSE community* as a whole.

### *Overview of Survey Questions*

First, to get a perspective of the survey respondents' overall view of software engineering knowledge within the CSE community, we begun the survey with some high-level questions:

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

Next, because our previous experience led us to believe that most CSE developers lack formal software engineering training, we asked the respondents to rank the following choices from 1 to 4 relative to how they obtained their software skills (4 meaning most used and 1 meaning least used):

- reading books
- attending training courses
- co-workers
- learned on my own

Next, the bulk of the survey asked about specific software engineering practices commonly used in the commercial software development community:

- Software Lifecycles
- Documentation
- Requirements
- Basic Design
- Intermediate Design
- V&V (Verification & Validation)
- Unit Testing
- Integration Testing
- Acceptance Testing
- Regression Testing
- Version Control/Change Management
- Issue/Bug Tracking
- Test-Driven Development
- Structured Refactoring
- Code Reviews
- Agile Methods

We chose these specific practices because, based on our experience, we believed them to be potentially useful for CSE developers and we expected that at least some of them were already in use within the CSE community. For each practice, the survey respondents rated each of the following items using a 5-point Likert scale (none, low, medium, high, very high):

- relevance to my work

- personal level of use
- personal familiarity
- team level of use
- team's familiarity

Finally, we gathered demographic information on the survey respondents. This data included: the type of institution in which they work (government lab, university, private company, other), number of years of CSE development, fraction of CSE software development that is research software vs. production software, and their educational degrees.

*Who the survey was sent to*

Our goal was to target as large a representative CSE community as possible. After piloting the survey locally, we sent it out to a number of CSE mailing lists, including several internal Sandia National Labs lists (Charon, Alegra, SIERRA, Xyce, Dakota), the Trilinos users and developers lists (trilinos.sandia.gov), the PETSc users and developers lists (www.mcs.anl.gov/petsc/petsc-as), the Consortium for Advanced Simulation of Light Waters Reactors (CASL) members list (www.casl.gov), and the Numerical Analysis Digest mail list (www.netlib.org/na-net).

*Demographics of respondents*

The 141 survey respondents were somewhat diverse. Most (82%) held a Ph.D. while 16% had a Master's degree. As shown in Figure 1, the three most common fields in which respondents

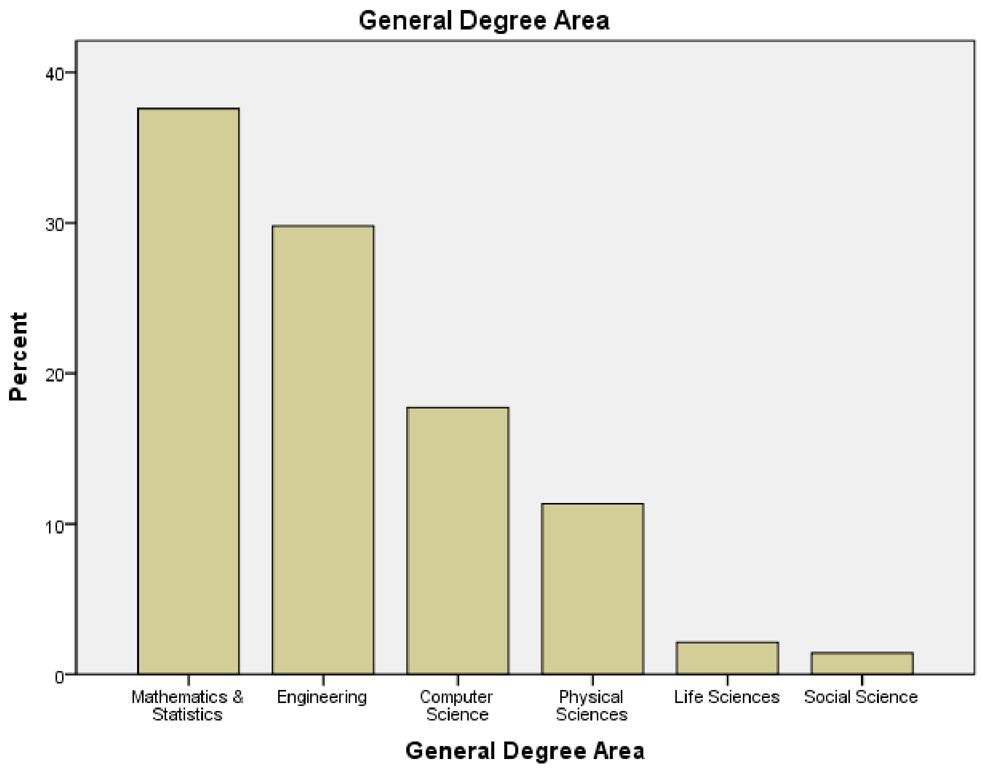

Figure 1 – General Degree Area

earned their highest degree were Mathematics & Statistics, Engineering, and Computer Science. About half (45%) of the respondents have been developing CSE software for between 3 and 14 years and a third (35%) for 15 to 26 years. About half the respondents (46%) worked at a university, a third (34%) worked at a government laboratory and the rest worked for private companies.

## Observations from the data

Our main question was: *Do CSE developers think they know enough about software engineering to allow them to produce high credibility CSE software?* Generally, the respondents were confident in their own level of SE knowledge: 92% indicated that their SE knowledge was sufficient for their own projects. The respondents were less confident about the SE knowledge of their peers: 79% said their team's SE knowledge was sufficient, and only 63% said that the SE knowledge of the CSE community in general was sufficient. As justification for these ratings, the respondents overwhelmingly indicated that their responses were based on their own or their team's experience in the field.

In response to the question of how the developers obtained their software engineering skills, the respondents rated "Learned on My Own" highest on a 1-4 scale (mean: 3.4), followed by "Reading Books" (2.8), "Co-workers" (2.5) and, finally, "Attending Training Courses" (1.8). This response also indicates that the primary method for acquiring software skill is through experience in the field.

Our knowledge of the CSE community led us to believe that within that community, there were two general types of software. *Research software* is software that is written with the primary goal of publishing a paper. *Production software* is written with the primary goal of producing software for real users. Prior to the survey, we suspected that the software engineering practices used in research-oriented software development would be different than those used in production-oriented software development. Research-oriented CSE software would likely be similar to prototyping while production-oriented CSE software would be more similar to traditional commercial software development with external users and their attendant needs (e.g., documentation, tracking externally reported bugs and feature requests).Therefore, we expected that developers who write research-oriented software would respond to the survey questions differently from those who write "production" software.

To better understand this factor, we asked: *"What fraction of your CSE software development effort is devoted to developing production software (main goal is to produce software for real users) versus developing other types of software, i.e. research software (main goal is to publish papers)"*. Figure 2 shows the distribution of responses across our sample with 0/10 indicating all time spent on research software and 10/10 indicating all time spent on production software. We were surprised to see a bimodal distribution: respondents tended to spend the majority of their time doing either mostly research or mostly production software development, with only a few splitting their time evenly across research and production software. Based on this data, we split the sample into three groups and analyzed the responses for each group independently:

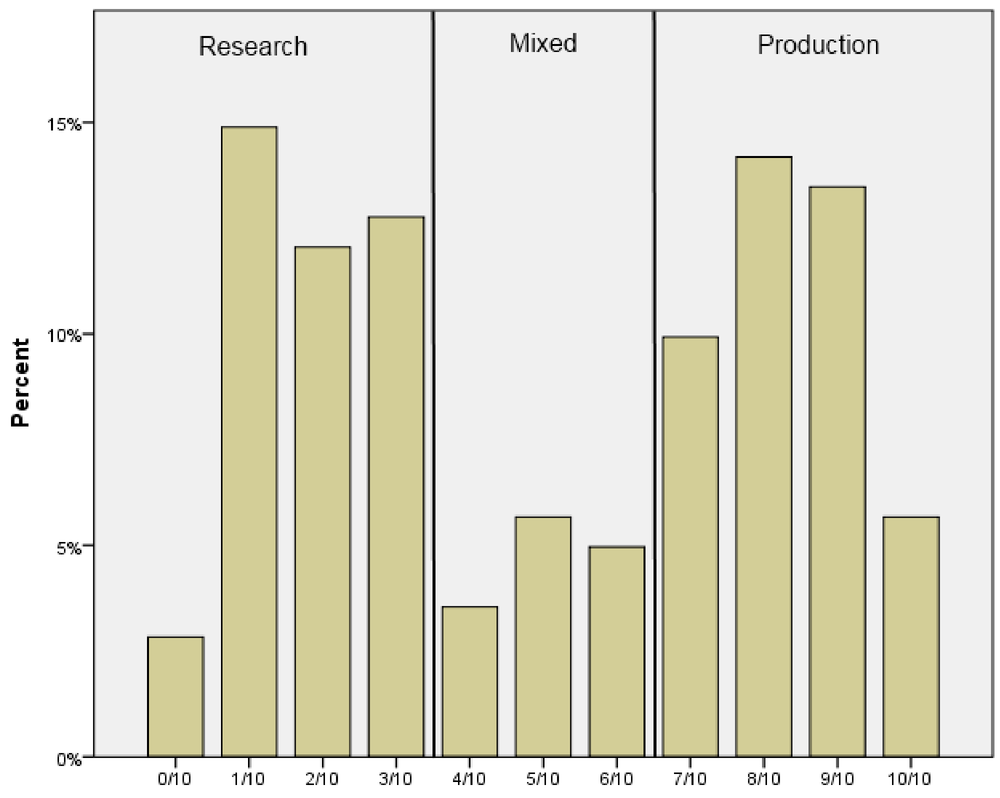

Figure 2 – Fraction of Effort Devoted to Production-Oriented Software

1. *Production software*: respondents who spent at least 70% of their time writing production software
2. *Mixed software*: respondents who spent between 30% and 70% of their time writing production software
3. *Research software*: respondents who spent no more than 30% of their time writing production software

For each software engineering practice listed earlier, we computed the average of the responses given by those in each of the three groups. Figure 3 shows the average response for each practice in terms of *Relevance to my work*, *Personal level of use*, and *Personal familiarity*, where 1 corresponds to *none* and 5 corresponds to *very high*. (We did not include similar graphs for Team Level of Use and Team Familiarity due to space. But, they looked similar to the graphs for Personal Level of Use and Personal Familiarity, with the average responses shifted to the left).

While the average scores for the production software development group were (almost) always higher than the average scores for the mixed group, which were always higher than the average for the research group, the gap between the production group's scores and the research group's scores varied substantially across practices. The mixed group's score was sometimes closer to

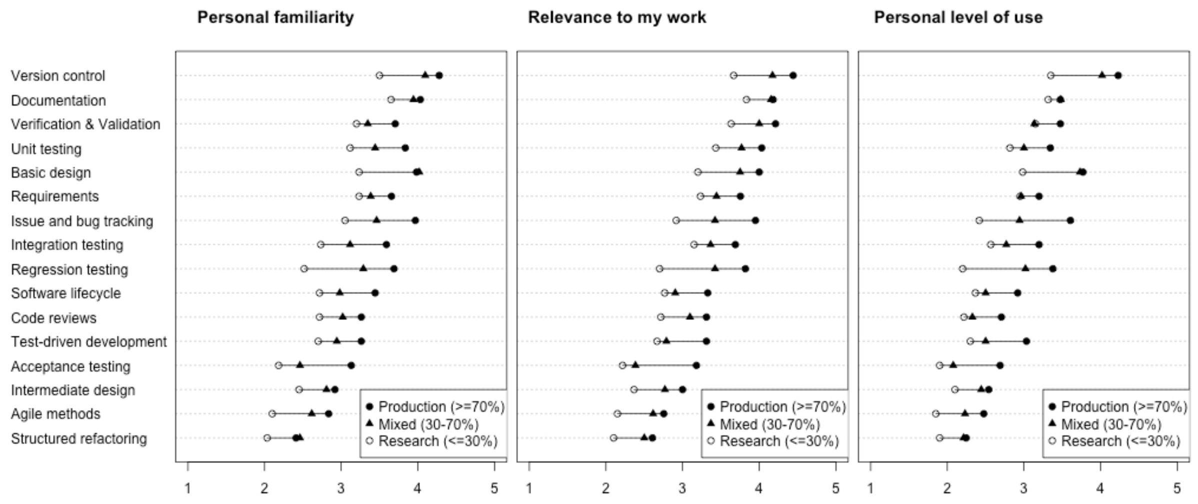

Figure 3: Average of the Responses for Each Software Engineering Practice

the production group and other times it was closer to the research group. The practices viewed as most relevant overall were Version Control, Documentation, and Verification & Validation.

There were a number of practices that exhibited a large variation between the research-oriented group and the production-oriented group, including: *Issue & Bug Tracking*, *Regression Testing*, and *Acceptance Testing*. In hindsight, it is not too surprising that such large differences were present for these practices. In terms of Issue & Bug Tracking, because research-oriented software is often short-lived and used by its developers, it is not as important for those developers to formally track bugs. Similarly, because research-oriented software is not meant for widespread external usage, conducting acceptance testing is not relevant. Finally, in terms of regression testing, production-oriented developers need to ensure that yesterday's features still work today, while a research-oriented developer may regard their software more as disposable prototypes and be less concerned about future usage.

On the other hand, there was very little difference between the production-oriented developers and the research-oriented developers relative to *Documentation,* with all three groups rating the topic between moderate and high. This result is surprising, given that production-oriented developers are more likely than research-oriented developers to create software for an external audience. Typically the need for documentation is driven, at least in part, by the presence of external users. Therefore, it is surprising that production-oriented developers did not view documentation as more important than research-oriented developers. Another potential explanation of this observation is that the definition of documentation might be different among research-oriented developers than it is among production-oriented developers. The level of documentation that may be sufficient for a research-oriented developer might be perceived as insufficient by a production-oriented developer.

## Conclusions

Based on these observations, we can draw a number of conclusions. One important caveat to remember regarding all the following conclusions is that all of the data reported in this survey was the personal opinions of the respondents and may or may not accurately reflect reality. These responses about various software engineering practices constitute a kind of self-assessment. However, an independent assessment of the true level of knowledge and use of the software engineering practices by the survey respondents and their associated CSE projects would likely change the results, although we are not sure what change would occur.

The results of the survey seem to verify many of our personal observations and reports from a prior survey [9]. CSE practitioners have very little formal software engineering training and tend to be mostly self-taught (either through personal experience or through some reading on their own). The survey respondents overwhelmingly believed that their software engineering knowledge and skills were at least "mostly sufficient" to achieve the goals of their CSE projects. Only 8% of respondents felt that they did not have sufficient software engineering knowledge and skills to achieve the goals of their CSE projects. However, the personal level of knowledge and use of many modern best practices (e.g. code reviews and refactoring) is relatively low. This discrepancy between the respondents' rating of their overall software engineering knowledge as at least "mostly sufficient" and their ratings of the individual software engineering practices shows that they do not know those practices. This discrepancy seems to support our hypothesis that the majority of individuals in the CSE community "don't know what they don't know." If true, this bias affects all of the responses.

Another telling result is that 37% of the respondents thought that overall the CSE community's skills were not adequate to advance the CSE field. This result would appear to constitute a serious self-diagnosed problem within the CSE community. But also consider the hypothesis that motivated this survey that most individuals in CSE "don't know what they don't know" about critical software engineering issues. This hypothesis would seem to suggest that the true level of critical software engineering skills and knowledge in the CSE community may actually be lower than reported in the survey. Some of the evidence for our hypothesis is evident in the responses with respect to perceived relevance and usage of various practices described below.

For all of the specific practices covered in the survey, *personal familiarity* was higher than *personal use*, *team familiarity*, and *team level of use*. Across practices, *relevance* was almost always rated higher than *familiarity*, or *level of use.* This observation suggests two conclusions: 1) CSE developers are not being forced to use practices they are not familiar with or do not think are important; and 2) CSE developers are not using all practices that they find relevant.

The low scores for *Intermediate Design* and *Structured Refactoring* suggest a potential explanation for why many larger CSE code projects are difficult to maintain. These two practices work together to control the accumulation of complexity in large-scale codes. The lack of use of these practices suggests that many of these CSE codes are not being sufficiently designed and without the use of *Structured refactoring* it is difficult to maintain reasonable conceptual integrity of the code over the (typically very long) life of modification and maintenance. Over time, the lack of (re)design and refactoring results in large, complex, fragile codes that become difficult to augment with new functionality once they reach a certain level of complexity.

Another major warning sign is the low scores for *Code reviews* and for *Agile methods*, suggesting that collaborative development practices [10] which are helpful for achieving sufficient quality in many situations are not being used. The lack of use of these practices suggests (and could explain) the relatively low quality of CSE software projects (at least in the early stages of major development).  Only after new feature development mostly ends on a complex CSE code is it that users begin to really test the code and revel the defects. Then, the code quality is greatly improved. In essence, this approach is simply high-volume beta testing [10].  In fact, we can argue (and support that argument with the responses in this survey) that really the only effective defect remove approach commonly used in most CSE projects is high-volume beta testing.  This situation explains why many CSE projects are so reticent to any code changes [4]. Code verification is primarily done by manual user testing and not by automated acceptance or regression testing (which are needed to catch new defects during development).

Finally, the fact that the respondents rated their knowledge of and the relevance of *Agile Methods* so low is quite interesting because Agile Methods are the standard software engineering process that most closely approximates what most CSE teams actually use [11]. Interestingly *Requirements* rated much higher than *Agile Methods*. It seems that if any requirements management is being done in the typical CSE project, it is mostly likely done in an iterative, Agile-like way. *Requirements Management* typically implies a large, waterfall-like, up-front requirements gathering phase. There is little evidence that most CSE teams operate that way. These results suggest a lack of training about appropriate software process management models or even exposure to these models in the respondents' organizations and projects.

When it comes to software development skills in the CSE community, perception is leading reality. If scientists and engineers continue to remain unaware that likely better practices of writing software are already available, the overall quality of CSE is unlikely to improve.

For future research, we plan to collect empirical data on the implications of the current state of CSE software engineering: anecdotal evidence suggests that many scientific codes are in a poor state, but there has been little systematic research to assess the state of CSE code quality against reasonably applicable best practices and standards from the broader software engineering community. We also plan to evaluate the hypothesis that there are cultural barriers to improving code quality through better software engineering practices. We have encountered CSE developers who assert that software engineering is simply common sense. To them, software engineering practices are merely crutches that are only useful for lesser mortals who develop software in the IT community (and lack PhDs from prestigious institutions). If it turns out that such attitudes are widespread in the CSE community, then we will need to do more than increase awareness about good software engineering: we will need to change the culture.

## References

[1] Z. Merali, "Computational science: ...Error ... why scientific programming does not compute," *Nature,* vol. 467, pp. 775-777, 12 Oct. 2010, 2010.

[2] Newsnight, "CRU's programming 'below commercial standards'," Dec. 4, 2009.

[3] H. Devlin, "Science blogger finds errors in Met Office climate change records," *The Times,* Feb. 16, 2010, 2010.

[4] S. M. Easterbrook and T. C. Johns, "Engineering the Software for Understanding Climate Change," *Computing in Science & Engineering,* vol. 11, pp. 65-74, 2009.

[5] L. Hochstein and V. R. Basili, "The ASC-Alliance Projects: A Case Study of Large-Scale Parallel Scientific Code Development," *Computer,* vol. 41, pp. 50-58, 2008.

[6] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull and M. V. Zelkowitz, "Understanding the High-Performance-Computing Community: A Software Engineer's Perspective," *IEEE Software,* vol. 25, pp. 29-36, 2008.

[7] R. Kendall, J. C. Carver, D. Fisher, D. Henderson, A. Mark, D. Post, C. E. Rhoades Jr and S. Squires, "Development of a Weather Forecasting Code: A Case Study," *IEEE Software,* vol. 25, pp. 59-65, 2008.

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

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

[10] S. McConnell, *Code Complete.* Microsoft Press, 2004.

[11] M. Poppendieck and T. Poppendieck, *Implementing Lean Software Development.* Addison Wesley, 2007.