The Relationship between Development Problems and Use of Software Engineering Practices in Computational Science & Engineering: A Survey

Dustin Heaton*, Jeffrey C. Carver*, Roscoe Bartlett†, Kimberly Oakes* and Lorin Hochstein‡
*University of Alabama, Tuscaloosa, AL, USA
†Oak Ridge National Laboratory, Oak Ridge, TN, USA
‡Nimbus Services, Arlington, VA, USA
dwheaton@ua.edu, carver@cs.ua.edu, bartlettra@ornl.gov, kloakes@ua.edu, lorin@nimbusservices.com

Abstract—**The development of software has become critical to progress in many important scientific and engineering fields. In general, the use of business/IT software engineering practices in these fields is relatively low. This paper describes the results of a survey of Computational Science & Engineering (CSE) developers that analyzed the current state of software engineering in the CSE community. Specifically, we examined four important CSE software development problems and the use of software engineering practices that should help address those problems. The results showed that in general, CSE developers are not using the software engineering practices that would most help those development problems as frequently as they could be.**

I. Introduction

Scientific and engineering breakthroughs in fields such as climate modeling, weather forecasting, high-energy physics, and cancer research, are increasingly enabled by the development and use of software. This software is often developed to replace dangerous or expensive physical experimentation and to aid in processing very large amounts of data. The development and use of this software is referred to collectively as Computational Science & Engineering (CSE). Because many critical decisions are based, at least partially, upon the output of CSE software, it is of utmost importance for this software to be correctly designed and implemented. Low software quality reduces the level of confidence that researchers and decision-makers can place on the results. In addition, due to many factors, including its complexity, the development of CSE software is effort-intensive.

Because a scientific or engineering problem must be of sufficiently complexity to require the development of software to support its investigation, developers often need advanced technical training, likely a PhD, in the domain to even understand the problem. As a result, because it is difficult for software engineers to understand the domain, scientists and engineers, who generally do not have a formal software engineering (SE) education, must also serve as the main developers of CSE software.

In the business/IT SE world, researchers and practitioners have developed practices to reduce development effort and increase software quality. Unfortunately, these practices are often not adopted by CSE developers [6]. A survey by Nguyen-Hoan et al. indicated that while there has been improvement in the adoption of SE practices by scientists, there is still a considerable amount of room for increased adoption [7]. It is likely that the low adoption rate of SE practices has an important effect on the resulting quality of CSE software.

As a result, researchers have conducted studies to identify factors that impact the adoption of SE practices by CSE developers. While there are some similarities between CSE software development and business/IT software development, there are also some important differences. Some of these characteristics suggest that SE practices need to be tailored for use in the CSE domain [8,14]. These characteristics include: CSE developers learn how to develop software from other CSE developers who also lack formal software engineering training [1]; many large CSE software packages were not initially designed to be large, but rather grew as a result of success [1]; CSE software is primarily used by its developer or its developers research group rather than primarily being used by external users [1]; CSE requirements gathering and discovery is difficult because the goal of the software is often to explore unknown domains to increase understanding rather than to solve a known, tractable problem [2–4,12]; verification and validation are difficult because often the expected result is not known or cannot be known *a priori* and there are multiple potential sources for defects [2,4,9,10]; and the scientific or engineering outcomes are viewed as being more important than choosing the most appropriate SE practices [2,4,11].

To investigate CSE developers adoption of SE practices, we conducted a survey of the CSE community. This paper describes the survey process and results and is organized as follows. Section II describes our prior survey that motivated the current survey. Section III describes the design of the current survey. Section IV discusses the data analysis process. Section V provides demographic data about the survey respondents. Section VI provides the main analysis results. Section VII interprets these results. Section VIII explains the threats to validity. Finally, Section IX concludes the paper and discusses future work.

II. Previous Survey

Prior to this survey, we performed a survey to judge CSE developers perception and use of SE practices [3]. Here we highlight some of the most interesting survey results that motivated the development of the current survey.

When asked about their level of general SE knowledge (i.e. not about any specific software engineering practices), 92% of the respondents indicated their knowledge was sufficient for their current projects. Only 78% indicated that their teammates level of knowledge was sufficient for their current projects (78%). Finally, only 63% believed that the CSE community in general had sufficient knowledge for their current projects.

To better understand of the respondents familiarity with specific SE practices, we listed 15 common SE practices and asked them to indicate whether they were familiar with a particular practice, whether the practice was relevant to their work and whether they used the practice. The 15 practices were: *Software Lifecycles/Software Process*, *Documentation*, *Requirements*, *High-Level Design/Architecture*, *Low-Level Design*, *Verification and Validation*, *Unit Testing*, *Integration Testing*, *Regression Testing*, *Version Control/Change Management*, *Issue/Bug Tracking*, *Structured Refactoring*, *Test-Driven Development*, *Code Reviews*, and *Agile Methods*. The respondents rated *version control*, *documentation*, and *verification & validation* as the most relevant practices. They rated *refactoring*, *agile methods* and *intermediate design* as the least relevant practices.

Another interesting result was that often respondents rated the relevance of a practice higher than they rated their use of that practice. This result suggests that for some reason developers were not using practices which they thought were relevant. Unfortunately, we did not anticipate this result so the survey did not include questions that would have provided insight into the source of this result.

The survey also asked the respondents to list any CSE software development problems that they believed could be addressed by adopting SE practices. This open-ended question was designed to gather as much information as possible without constraining the set of answers that could be given. A qualitative analysis of the data revealed four common problems: *rework*, *performance issues*, *regression errors*, and *forgetting to fix bugs that were not tracked*.

The results of this survey indicated the need to better understand the problems faced by CSE developers and why they were not using the SE practices they viewed as relevant. To gain this understanding, we developed a second survey, which is reported in detail in remainder of this paper.

III. Survey Design

To develop a more complete picture of CSE developers perception of their knowledge and use of SE practices, we conducted a follow-on survey. To more accurately characterize the state of the CSE community, this survey included some questions from the original survey plus new questions to address expand upon the results from the original survey. Here we list the motivation for some specific additions to the survey and describe the target audience.

A. Changes To Original Survey

First, the original survey did not define each of the SE topics, because we assumed that respondents would be familiar with them. After discussing the survey results with CSE experts, we determined that it was important to determine whether survey respondents agreed with the standard SE definitions. This survey gave a common definition for each term and asked survey respondents whether they agreed with this definition. If they disagreed, they could provide their own definition. This information provides us with a better understanding of level of SE knowledge of CSE developers.

Second, in the original survey, the open-ended question about the most common software development problems faced by CSE developers revealed four common problems. To better understand how universal those problems were, the second survey asked the respondents to indicate the frequency and severity of each problem.

Third, similar to the most common software development problems, the original survey respondents reported barriers to the adoption of SE practices. To better understand how universal these barriers were, the second survey asked respondents to rate the importance of the top four barriers identified by the first survey.

Finally, the original survey did not gather the primary programming languages used by the respondents. The choice of programming language may affect the usage of some SE practices. So, the second survey gathered information about the use of programming languages.

The complete survey is available at: http://dheaton.students.cs.ua.edu/SECSE/survey2.htm.

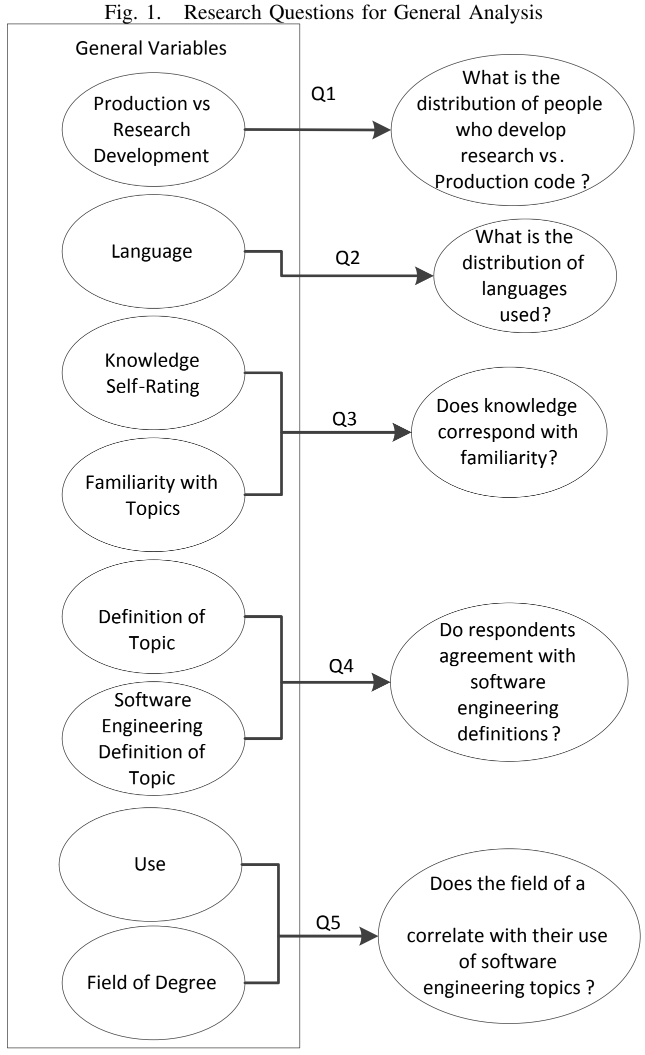
B. Target Audience

This survey targeted a broader subset of the CSE community than the first survey. To get a representative sample, we sent the survey to a number of CSE mailing lists including: several internal Sandia National Labs lists, the Trilinos users and developers lists, the PETSc users and developers lists, the Consortium for Advanced Simulation of Light Waters Reactors members list, the Numerical Analysis Digest mail list, the CSGF Alumni List, the NERSC list, the NCCS list, the SciComp and CSMD lists at Oak Ridge National Laboratory, the NICS list, the NEAMS list, HPC-Announce, and the SIAM CSE list. In addition, we sent the survey to the Computational Scientists & Engineers, Scientific Discovery through Advanced Computing, and Computational Science LinkedIn groups. We received a total of 151 survey responses.

IV. Analysis Process

While most of the survey data is quantitative, a significant portion of the data is qualitative. To analyze the qualitative survey responses, we performed a qualitative analysis process that including coding and classifying the responses to make them easier to analyze.

Fig. 1. Research Questions for General Analysis


General Variables
Production vs Research Development
Q1
What is the distribution of people who develop research vs. Production code ?
Language
Q2
What is the distribution of languages used?
Knowledge Self-Rating
Familiarity with Topics
Q3
Does knowledge correspond with familiarity?
Definition of Topic
Software Engineering Definition of Topic
Q4
Do respondents agreement with software engineering definitions ?
Use
Field of Degree
Q5
Does the field of a
correlate with their use of software engineering topics ?

First, to provide a more concise view of the responses, one author extracted the main ideas in each qualitative answer. This analysis focused on three primary questions: why the respondents used the chosen languages, why the respondent disagreed with our definitions, and why the respondent rated the relevance of a SE practice differently from their use of that practice.

Using this information, another author proposed an initial coding scheme for each question by grouping related answers. Two additional authors then examined the coding schemes to agree upon a final coding scheme for each question. Then the two authors who performed the initial analysis and coding scheme individually coded the answers to each question using the finalized coding schemes. After they had both completed their initial coding, they compared their results. In general, the codings were similar. In most cases where the codes differed, the two coders resolved the conflicts through discussion. Lastly, one of the other authors met with the two coders to resolve any unresolved coding conflicts. This process provided us with confidence in the results of the coding process.

V. Demograhics/General Analysis

This section provides an overview of general demographic information about the sample. We used this data to answer five research questions, which are illustrated in Figure 1:

1) What is the distribution of people who develop research vs. production code?
2) What is the distribution of languages used?
3) Does a developers view of their own knowledge correspond to their familiarity with individual SE practices?
4) Do the respondents agree with the standard definitions of the SE practices given on the survey?
5) Is there a relationship between a respondents highest degree and their usage of SE practices?

A. Research vs. Production Code

Based on our discussions with members of the CSE community and our own collective experience working with CSE projects, we believed that within the CSE community there are two types of software. *Research software* is developed primarily for use by the developer or the developers research group and has a primary goal of gaining scientific or engineering insight to publish a technical paper. Conversely, *production software* is developed with the primary goal of producing software for external end-users. Because of the difference in goal and focus, it is likely that the developers of these types of software will behave differently from each other.

The survey asked the respondents to report the percentage (in blocks of 10%) of time they spend developing software for external users (i.e. production software). Based on this information we group those who spent 30% or less of their time on software for external users as *Research Developers*, those who spent 70% or more of their time on software for external users as *Production Developers*, and the rest as *Mixed Developers*. In the first survey, the respondents represented a bimodal distribution with the bulk of respondents being either *Research Developers* or *Production Developers* with very few *Mixed Developers*. However, as Figure 2 shows, the respondents to this survey were more randomly distributed.

B. Language Distribution

Prior to the survey, we hypothesized that most respondents would use FORTRAN and that few would use high-level languages such as JAVA. Figure 3 shows that FORTRAN was not as dominate as we hypothesized. While FORTRAN was popular among the survey respondents (used by 53%), C++ was even more popular (62%).

C. General Knowledge vs. Familiarity with Specific Topics

The previous survey showed a strong relationship between a developers view of their overall SE knowledge and their familiarity with individual SE practices. However, in this survey, a developers view of their own overall SE knowledge appeared to be unrelated to their familiarity with individual SE practices. We conducted a Chi-Square analysis to determine whether survey respondents who rated their overall knowledge high were also familiar with the individual SE practices.

Fig. 2. Distribution of Developers between Research and Production Types

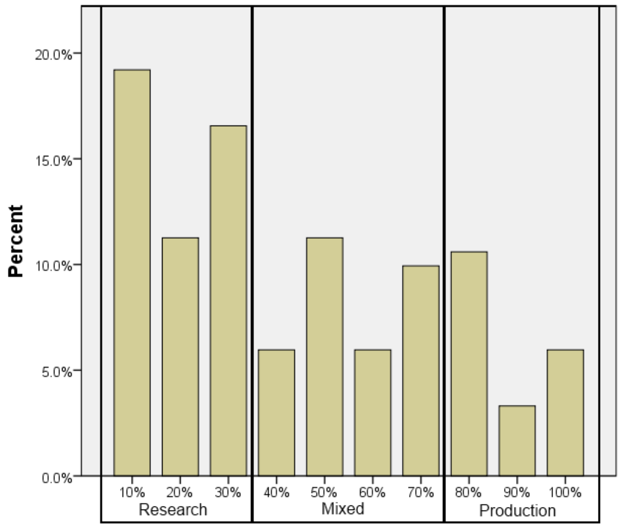
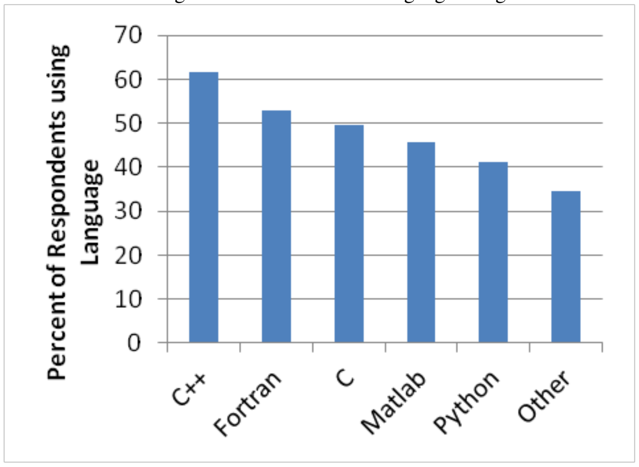


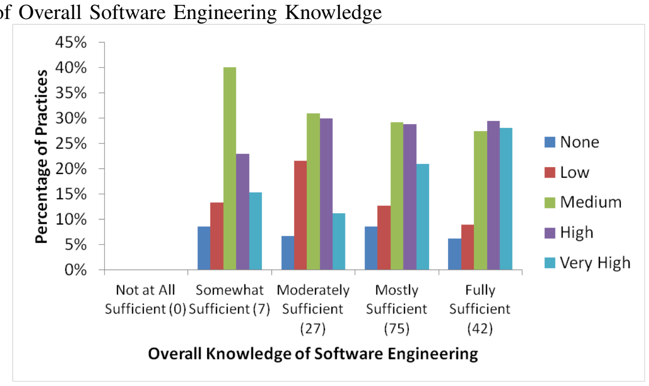
Fig. 3. Distribution of Language Usage



The results of this analysis showed no significant correlation between the distributions.

Figure 4 shows this data graphically. Each respondent rated their familiarity with 15 SE practices. The categories on the x-axis represent the respondents overall rating of their SE knowledge. The number in parenthesis is the number of respondents who gave that overall rating. The clustered bars represent the percentage of the 15 SE practices that respondents rated at each level of familiarity. For example, 42 respondents indicated that their general SE knowledge was *fully sufficient*.Those 42 people rated their level of familiarity with the various SE practices as very high 28% of the time, high 29% of the time, medium 27% percent of the time, low 9% of the time, and none 6% of the time. For the respondents who considered their knowledge of SE to be either *moderately*, *mostly*, or *fully* sufficient the number of SE topics of which they considered themselves as having either medium or high knowledge is approximately equal. Interestingly, the respondents who considered their knowledge of SE to be only somewhat sufficient rated themselves as having only a medium familiarity more often.

Fig. 4. Familiarity with Software Engineering Topics Grouped by Self-rating of Overall Software Engineering Knowledge



Both the statistical and the graphical results indicate that in general a respondents rating of their overall SE knowledge is unrelated to their level of familiarity with the list of 15 SE practices on the survey. Therefore, either the respondents overestimated their general SE ability or there are other practices with which they are more familiar.

D. Agreement with Definitions

Based on our discussions with members of the CSE community, we hypothesized that a large number of respondents would disagree with the definitions of the individual SE practices based on different interpretations of the same term in different communities. Instead, we found that, for almost every SE practice, less than 5% of the respondents disagreed with the given definition. The two exceptions to this result were the *software lifecycles* and *agile methods* practices with 20.5% and 11.9% disagreement respectively. A potential explanation for this high level of agreement is that respondents did not want to explain why they disagreed. We do not think this explanation is valid because, while we asked for a reason, we did not require one to be given.

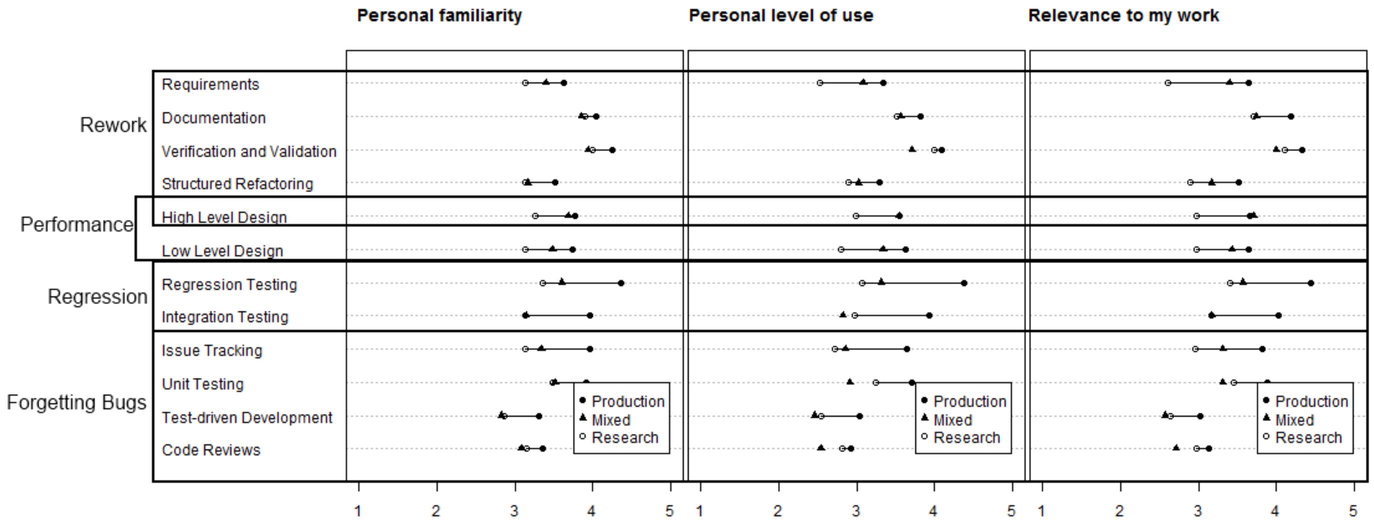
E. Highest Degree Earned vs. Usage of SE Practices

We hypothesized that there would be a significant relationship between the field in which a respondent holds their highest degree and their usage of SE practices. We further hypothesized that respondents whose highest degree was in Computer Science would be more likely to use SE practices than the other respondents. Relative to the first hypothesis, a Chi-Square test showed no significant relationship between a respondents academic field and their level of use of SE practices. The second hypothesis also proved false as the largest percentage of respondents who rated their use of SE practices as *high* or *very high* were engineers.

VI. Analysis

To better understand how to help improve the quality of CSE software, the remainder of this paper discusses the four most common problems CSE software developers face: *rework*, *performance issues*, *regression errors* and *forgetting to fix bugs because they are not tracked*. These problems originated on our first survey and were verified on the current

Fig. 5. Mean Ratings of Software Engineering Topics grouped by Problem Addressed



survey. On the current survey we asked respondents to rate the frequency and severity of each problem on a five-point scale. The respondents indicated that all four problems occur with a higher than average frequency and are of higher than average severity. Therefore, all four problems are important problems that should be studied further.

We also asked the survey respondents to indicate which of the four problems they would eliminate if they could eliminate only one. Based on the answers to that question, we sort the problems in descending order of importance: *rework*, *performance issues*, *regression errors*, and *forgetting to fix bugs because they are not tracked*. The remainder of this section discusses each problem in more detail.

Each problem could be addressed by the use of some of the SE practices covered by the survey. So, for each problem discussed below, we first propose and justify a set of SE practices that should be most relevant to alleviating the problem. Then, we discuss the respondents level of familiarity, relevance and use of those practices. The next section provides a more detailed analysis of what these results mean for CSE software development.

Figure 5 provides an overview of the data discussed in the following subsection. The figure lists the SE practices on the y-axis, grouped by problem areas. Note that *High-Level Design* is relevant to both *Rework* and *Performance*. For each topic, the figure shows the average response for *Personal Familiarity*, *Personal Level of Use* and *Personal Relevance to Work*. The averages are shown separately for the **Production Developers**, **Research Developers** and **Mixed Developers**. Note that in all cases, the production developers rated the practices as more relevant and more used than the research developers did. This result is not completely surprising given that production developers are writing software that is more like business/IT software because it will be used by external users.

A. Rework

Rework was the most frequent problem reported by the respondents, which is not surprising because it is a major problem in all software development disciplines. Therefore, five SE practices on this survey should either reduce the need for rework or reduce the effort required to perform rework: *requirements*, *verification & validation*, *high-level design*, *documentation*, and *structured refactoring*.

Proper usage of *requirements* helps to reduce the need for rework by determining what the software needs to do prior to implementation, thereby reducing the chances of unanticipated changes. Additionally, in an agile environment, like most CSE projects, the proper use of *requirements* helps to document and manage evolving requirements. The process of creating and using requirement documents is currently used at a moderate level by **production developers** but is only used at a low level by **research developers**. Similarly, **research developers** find this topic to have low relevance to their work while **production developers** find it to have high relevance.

Various types of *verification & validation* performed throughout the development process decrease the likelihood of rework. The more problems that can be identified and removed during development, the less that will have to be fixed via rework. Both **research developers** and **production developers** claimed a high or very high level of relevance and use for *verification &validation*.

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

Unlike the previous three practices (other than *requirements*), proper *documentation* does not necessarily prevent the need for rework. The presence of complete and accurate *documentation* can significantly lower the amount of effort required to perform rework. *Documentation* is, much like *requirements*, used more heavily by **production developers**, but in this case **research developers** also agree that it is highly relevant and use it frequently.

Similarly *structured refactoring* both prevents the need for in-depth rework and reduces the effort required to perform rework. It reduces the need for rework by allowing developers to fix design deficiencies discovered after the design is finalized without changing the functionality of the software. It can reduce the effort required to perform rework if the refactoring is done to reduce complexity and increase maintainability. As with *verification &validation*, **production developers** reported only a slightly higher level of usage of *structured refactoring* than did **research developers**. However, **production developers** considered *structured refactoring* to be significantly more relevant than did **research developers**($\chi^2_{df} = 4, p = .05$).

B. Performance Issues

Performance issues were the second most frequently reported problem. However, performance issues were considered to be somewhat less severe than the other problems. The two SE practices on our survey that most directly relate to improving performance (i.e. speed of execution) are *high-level design* and *low-level design*. In both cases correct design decisions can either optimize performance or slow down performance.

High-level design, or architecture, affects global performance by determining how the system is organized and what types of communication must occur. As discussed in the previous subsection, **production developers** viewed *high-level design* as being more relevant than did **research developers**.

Low-level design focuses more on optimizing the performance within a module through use of appropriate data structures and algorithms. **Production developers** reported a high rate of usage of *low-level design* while **research developers** only reported a low to medium level of use. Similarly, **research developers** reported a medium level of relevance while **production developers** reported high relevance.

C. Regression Errors

Regression errors were the third most frequently reported problem. **Production developers** found regression errors to be more problematic than did **research developers**, but there was little reported difference in the frequency of regression errors. Two types of testing can help to address regression errors: *regression testing* and *integration testing*.

The goal of *regression testing* is to detect regression errors. Therefore this practice is clearly the most relevant to the problem. The usage of *regression testing* by **research developers** is fairly evenly distributed across the scale of low to high usage, but **production developers** report a very high level of use. **Research developers** further tend to view *regression testing* as having a high level of relevance with most **production developers** rating it as very highly relevant.

While it is not specifically intended to detect regression errors, *integration testing* can be used to verify that no new regression errors have entered the software when new code is added. Much like *regression testing*, *integration testing* was used only by some **research developers**. In addition, there was no real pattern in the **research developers** views of the relevance of *integration testing*. **Production developers**, however, reported a high level of both usage and relevance.

D. Forgetting to Fix Bugs Because They are not Tracked

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

Using *issue/bug tracking* software allows CSE developers to track bugs with a minimal amount of extra work. Despite the similar distributions for frequency and severity of this problem, there was a strong divide between the use of *issue/bug tracking* software between **research developers** and **production developers**. **Research developers** reported a low level of use and **production developers** reported a high level of use. The same divide occurred with regards to the respondents views of the relevance of *issue/bug* tracking software.

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

Code reviews are important because they provide a chance to find and fix mistakes that were overlooked in the design phase. Similar to *unit testing*, there was no significant relationship between the usage and relevance of code reviews and type of developer (**research** or **production**). The distribution of answers for usage and relevance for each type of developers was approximately normal.

Test-driven development is a development process that is particularly well-suited to the development of complex software. It forces the developer to add one piece of functionality at a time, simplifying the testing process. The results for the usage and relevance of *test-driven development*, however, were considerably different from the other results. While **Research developers** were as likely as **production developers** to use *test-driven development*, they viewed it as more relevant than did **production developers**.

VII. Discussion and Interpretation of Results

This section uses the qualitative survey data to interpret the results in the previous section in terms of their effects on the practice of CSE software development. The qualitative data used is drawn from the respondents comments about why they saw a topic as relevant but did not believe they were using it at a consistently high level.

A. Rework

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

The most common factor among the practices related to rework is that they require a lot of effort outside of programming. The benefit of this extra effort is perhaps not readily obvious. This idea is supported by a number of respondents who explained that their low levels of use for these practices resulted from limited people, resources, and time to perform them.

As the benefits of these practices are not readily obvious, the first step forward in addressing this problem is to better inform CSE developers making the decisions of where to spend resources about how utilizing these practices can reduce the need for expensive rework. Secondly, we must determine whether the existing SE techniques are feasible for CSE.

B. Performance Issues

Performance issues may be more related to hardware limitations as they are to software. Even so, both *high-level design* and *low-level design* can help with this problem. Many respondents who reported a low level of use of *high-level design* claimed that they had limited familiarity with it. Of those who believed they were familiar with *high-level design*, a large number saw it as having little relevance. Most of those who saw *high-level design* as having little relevance based this belief upon the fact that they were working either on small projects or with legacy code. The respondents stated that both situations force the developer to use a given architecture without being able to modify it. Many respondents also found the lack of a strong *high-level design* made it impossible for them to produce detailed *low-level design*. The amount of up-front effort required to produce *low-level design* hampered the ability of developers to utilize the practice. These observations indicate that there is a need for CSE developers to be better trained in *high level design*.

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

C. Regression Errors

Regression testing was seen as important by almost all of the respondents who commented on their usage of it. The reasons given for not utilizing *regression testing* were largely related to time constraints or a lack of familiarity with how to perform it. Another common issue, however, was that the developers did not have an automated *regression testing* tool for a large scientific problem.

The respondents also saw a considerable need for *integration testing* that was hindered by a few underlying problems. In many cases, developers did not have the support for or familiarity with *unit testing* needed to build integration tests. Otherwise, most respondents did not do the integration of software units themselves and they assumed that the developer who did the integration performed any needed tests. These problems would also make it difficult to implement *regression testing* because it relies upon an existing test suite for maximum efficiency. *Regression testing* also requires all developers involved with the software to use it.

Addressing these issues will require two steps. First, CSE developers need to be better trained in *unit testing*. Secondly, once they are able to perform unit tests, CSE developers will need automated regression testing tools developed to work specifically in the CSE environment.

D. Forgetting to Fix Bugs Because They are not Tracked

Both types of developers (**research** and **production**) viewed this problem as the most severe. Despite the importance of this problem, none of the respondents used a formal *issue/bug tracking* system. Instead they either used mailing lists or tried to keep track of bugs mentally. However, many of the respondents did believe that the adoption of such a system would be beneficial.

The respondents highlighted three main problems with *unit testing*:

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

A number of the respondents who claimed they did not use *test-driven development* unintentionally used an approach that could be adapted to *test-driven development*. They used an iterative method of adding a feature and then testing to be sure it worked properly. There were four major barriers to the adoption of *test-driven development*: not enough resources, lack of coherent high and low level designs, complicates the creation of test cases, and not familiar enough with the practice to implement it.

Use of *code reviews* was primarily limited by two factors. First, the developer was often not working in a large enough group to perform formal code reviews. The problem of group size is complicated by the fact that scientific developers are frequently exploring questions that are too complicated for external developers to understand the code. Second, many of the benefits of *code reviews* are not directly obvious (i.e. determining factors that cause bugs to enter code reduces the chances of similar bugs entering later codes). This lack of recognition of the benefits results in a development team giving the review process a lower priority than other tasks.

VIII. Threats to Validity

First, we have no way to know what motivated developers to respond to the survey. This fact means that it is possible that the respondents are not a representative sample of the CSE community and may be biased. However, the number of respondents compares favorably to a number of respondents in other surveys [7,13].

Second, although we asked the respondents to answer questions about the topics using the given definitions, we had no way to ensure that they actually did so. However, this threat is relatively minor because the majority of respondents agreed with the given definitions.

Third, another potential threat to validity shows itself in the language usage of the responders. It is generally believed that FORTRAN is the most heavily used language in CSE development. However, almost as many respondents in our sample used C and C++ as used FORTRAN. This result may indicate a bias in the type of people who chose to respond to the survey. Because we did not ask survey respondents which language was their primary language, we are unable to perform a detailed analysis about any effects these variables could have on the results. In future surveys we will address this issue.

IX. Conclusion and Future Work

This work shows that CSE developers who produce software for external users (i.e. production software) are both more likely to view any given problem as being frequent and severe and more likely to use relevant SE practices. The fact that these developers see the problems as more frequent and severe suggests that they may be more concerned with software quality than **research developers**. If the CSE developers who are most concerned with software quality can make use of SE, there is evidence that these SE practices are, in fact, applicable to the development of CSE software, something many in the CSE community argue against [5].

In addition, we found that many of the practices that should help address frequent and severe problems have not been adopted by a large number of the respondents. For these practices, the primary reason given for lack of adoption was that the respondent did not know how to use the practice itself or other practices that serve as a foundation. This result suggests that either CSE developers need more training or that the existing implementations of those practices are not suitable for the CSE domain and need to be tailored appropriately.

The fact that CSE developers who primarily write software for external users were already attempting to adopt SE practices to address specific problems and that CSE developers as a whole did not report a high level of use of these practices suggests that there is a high likelihood that CSE development can be enhanced by tailoring SE practices to better fit CSE development.

For future work, in order to determine how well SE practices actually address the CSE developers problems, we plan to conduct a series of case studies. In these case studies, we will work directly with individual CSE teams to evaluate the use of various practices on their projects so that we can help tailor those practices to work more effectively. These case studies will also help determine what changes need to be made to SE practices in so they can better fit the needs of CSE developers.

Acknowledgement

This work was largely supported by a NSF Research Experience for Undergraduate grant (CCF-0851824) and a Department of Education GAANN grant (P200A100182).

References

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

[2] J. Carver, R. 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*, may 2007, pp. 550 –559.

[3] J. C. Carver, R. Bartlett, D. Heaton, and L. Hochstein, "Self-perceptions about software engineering: A survey of scientists and engineers," Department of Computer Science, The University of Alabama, Tech. Rep. SERG-2011-04, Nov. 2011. [Online]. Available: http://software.eng.ua.edu/reports/SERG-2011-04

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

[5] M. A. Heroux and J. M. Willenbring, "Barely sufficient software engineering: 10 practices to improve your cse software," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, ser. SECSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 15–21.

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

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

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

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

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

[11] J. Segal, "Some problems of professional end user developers," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VLHCC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 111–118.

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

[13] F. Shull, M. G. Mendonca, V. Basili, J. Carver, J. C. Maldonado, S. Fabbri, G. H. Travassos, and M. C. Ferreira, "Knowledge-sharing issues in experimental software engineering," *Empirical Software Engineering*, vol. 9, pp. 111–137, 2004.

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