What We Have Learned About Using Software Engineering Practices in Scientific Software

Jeffrey Carver
University of Alabama
carver@cs.ua.edu

November 29, 2017

@SE4Science
Scientific Software Community

- Surveys
- Workshops
- Direct Interactions
- Case Studies
Community Surveys
Community Surveys: 
First Survey

• Sufficiency of SE Knowledge
  • Personally - 92% said yes
  • CSE community - 63% said yes

• Research vs. Production

• Reported 4 Key Problems
  • Rework
  • Performance issues
  • Regression
  • Forgetting to fix bugs not tracked
Community Surveys: Second Survey

• Broad subset of Computational Science audience – 151 responses

• Level of usage of various SE practices

• Generally agreed with our definitions of SE terminology
Community Surveys: Second Survey

Case Studies:

Goals

• Support scientific developers

• Gather information about effective and ineffective practices

• Understand and document software development practices

• Provide feedback to teams
<table>
<thead>
<tr>
<th></th>
<th>FALCON</th>
<th>HAWK</th>
<th>CONDOR</th>
<th>EAGLE</th>
<th>NENE</th>
<th>OSPREY</th>
<th>HARRIERS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Duration (Years)</strong></td>
<td>~ 10</td>
<td>~ 6</td>
<td>~ 20</td>
<td>~ 3</td>
<td>~ 25</td>
<td>~ 10</td>
<td>~ 8</td>
</tr>
<tr>
<td><strong># of Releases</strong></td>
<td>9 (production)</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>?</td>
<td>?</td>
<td>~ 16</td>
</tr>
<tr>
<td><strong>Staffing (FTEs)</strong></td>
<td>15</td>
<td>3</td>
<td>3-5</td>
<td>3</td>
<td>~10 (100’s of contributors)</td>
<td>~ 10</td>
<td>5 primary + students</td>
</tr>
<tr>
<td><strong>Customers</strong></td>
<td>&lt; 50</td>
<td>10s</td>
<td>100s</td>
<td>None</td>
<td>~ 100,000</td>
<td>100s</td>
<td>10s</td>
</tr>
<tr>
<td><strong>Code Size (LOC)</strong></td>
<td>~ 405,000</td>
<td>~ 134,000</td>
<td>~ 200,000</td>
<td>&lt; 100,000</td>
<td>750,000</td>
<td>150,000</td>
<td>50,000</td>
</tr>
<tr>
<td><strong>Primary Languages</strong></td>
<td>F77 (24%), C (12%)</td>
<td>C++ (67%), C (18%)</td>
<td>F77 (85%)</td>
<td>C++, Matlab</td>
<td>F77 (95%)</td>
<td>Fortran</td>
<td>C++ (50%), Python (50%)</td>
</tr>
<tr>
<td><strong>Other Languages</strong></td>
<td>F90, Python, Perl, ksh/csh/sh</td>
<td>Python, F90</td>
<td>F90, C, Slang</td>
<td>Java Libraries</td>
<td>C</td>
<td>C</td>
<td>None</td>
</tr>
<tr>
<td><strong>Target Hardware</strong></td>
<td>Parallel Supercomputer</td>
<td>Parallel Supercomputer</td>
<td>PCs to Parallel Supercomputer</td>
<td>Embedded Hardware</td>
<td>PCs to Parallel Supercomputer</td>
<td>Parallel Supercomputer</td>
<td>Linux, Windows</td>
</tr>
</tbody>
</table>

*Case Studies*
Case Studies
Lessons Learned
Lessons Learned:
Validation and Verification

http://dilbert.com/strip/2010-11-07
Lessons Learned: Validation and Verification

• Vary in formality and completeness
  • Core algorithms vs. User Interactions
  • Percentage of code tested
  • Dedicated testers vs. End users

• Required by sponsor?

• Existing verification techniques not useful

“V&V is very hard because it is hard to come up with good test cases”
Lessons Learned:
Validation and Verification

“I have tried to position CONDOR to the place where it is kind of like your trusty calculator – it is an easy tool to use. Unlike your calculator, it is only 90% accurate ... you have to understand that then answer you are going to get is going to have a certain level of uncertainty in it. The neat thing about it is that it is easy to get an answer in the general sense <to a very difficult problem>.”

“We have a rule of thumb. We plot 2 lines (from Matlab and C++ programs) and if close, then it is ok.”

“It is an engineering judgment as to which errors are important and which ones are on the margins”
Lessons Learned: Validation and Verification

• Implications
  • Traditional software testing methods are not sufficient
  • Need methods that ensure the quality and limits of software

• Suggestions
  • Inspections
  • Formal planning
  • Use of regression test suites
Lessons Learned: Development Goals

• Multiple goals are important
  • Performance – software is used on supercomputer
  • Portability and Maintainability – platforms change multiple times during a project

• Success of a project depends on the ability to port software to new machines

• Implications
  • The motivation for these projects may be different than for traditional IT projects
  • Methods must be chosen and tailored to align with the overall project goals
Lessons Learned:
Agile vs. Traditional Methodologies

I’ll need to know your requirements before I start to design the software.

First of all, what are you trying to accomplish?

I’m trying to make you design my software.

I mean what are you trying to accomplish with the software?

I won’t know what I can accomplish until you tell me what the software can do.

Try to get this concept through your thick skull: the software can do whatever I design it to do!

Can you design it to tell you my requirements?

Lessons Learned:
Agile vs. Traditional Methodologies

- Requirements constantly change as scientific knowledge evolves

- “Agile” software development methods
  - Tend to be more adaptable to change
  - Favor individuals and practices over process and tools

- Teams operate with agile philosophy by default

- Implications
  - Appropriate, flexible SE methodologies need to be employed for CSE software development
  - Agile-inspired approaches may be most appropriate
Lessons Learned:
Development Environments

http://dilbert.com/strip/1992-09-08
Lessons Learned: Development Environments

They all [the IDEs] try to impose a particular style of development on me and I am forced into a particular mode

• Developers prefer flexibility of the command line over an Integrated Development Environment (IDE)

• Developers believe that:
  • IDEs impose too much rigidity
  • They are more efficient typing than navigating menus

• Implications – developers do not adopt IDEs because:
  • They do not trust the IDE to automatically perform a task in the same way they would do it manually
  • They expect greater flexibility than is currently provided
  • Prefer to use what they know rather than change
SE4Science Workshops
SE4Science Workshop Series
http://SE4Science.org

• Facilitate interaction between SE and Computational Scientists

• Held at ICSE, ICCS, and SC

• Discussion Topics
  • Testing scientific software
  • Trade-offs between quality goals
  • Research Software vs. IT Software
  • Crossing the communication chasm
  • Measuring impact on scientific productivity
  • Reproducibility of results
SE4Science Workshop Series
Domain Characteristics

• Complex domains

• Main focus on science

• Long lifecycles

• Investigation of unknown introduces risk

• Unique characteristics of developers
  • Deep knowledge of domain – lack formal SE
  • Often the main users of the software
• Stakes not high enough to make testing important

• Needs differ across domains

• Focus on process transparency

• Guaranteed not to give an incorrect output
SE4Science Workshop Series
Crossing the Communication Chasm

• Need to eliminate the stigma associated with SE

• Software Engineers need to
  • Understand domain constraints
  • Understand specific problems
  • Learn from Computational developers
  • Describe SE concepts in terms familiar to Computational developers

• Need people with expertise in both SE & Computational Science

• Computational teams need:
  • To realize a problem before needing help
  • Real examples of SE success within their domain
SE4Science Workshop Series

Scientific Impact

• Need to evaluate impact

• Scientific productivity ≠ Software productivity

• Need results in a relatively short time
  • Self-assessments
  • Word of mouth
SE4Science Workshop Series
http://SE4Science.org

• Next edition – during ICSE’18

• Gothenberg, Sweden

• Please consider attending

http://SE4Science.org/workshops/
Direct Interactions
One Possible Methodology

1. Perform Case Study

2. Develop Software Engineering Techniques

3. Deploy and Evaluate

4. Synthesize Results

Strengths & Weaknesses in Development Process

Software Engineering Techniques

Project Team
Successful SE/CSE Interactions: TDD - Sandia

• Student spent semester at Sandia

• Taught and modeled TDD on a science code project

• Developed 2 tests for each PDE
  • Small number of steps
  • Whole time evolution

• Lessons Learned
  • Mitigated risks in changing requirements
  • Reduced developer effort
  • Continuous feedback from customer
Successful SE/CSE Interactions: TDD - Sandia

Successful SE/CSE Interactions: Peer Review - ORNL

- Student spent summer with science team at ORNL
- Taught team peer code review process
- Team adopted and continued on own
- Anecdotal Benefits
  - Found faults that would not have been found with traditional testing
  - Adopted coding standard for readability
Ongoing Work
“Bad By Admission” Code:

• Code that is actively recognized as deficient
  • Indicated by TODO or FIX
  • Often not fixed

• Compare Scientific and other software in GitHub
  • Compared 10 projects
  • Scientific code has 2x as many TODOs
Software Metrics in Scientific Software

• Survey of scientific software developers

• Goals
  • Understand knowledge and use of metrics
  • Understand perceived usefulness of metrics
  • Gain some insight into software process
Software Metrics in Scientific Software: Knowledge and Use of Metrics

**Knowledge**

- Very High: 0
- High: 5
- Average: 10
- Low: 15
- Very Low: 0

**Usefulness**

- All of the Time: 0
- Often: 5
- Sometimes: 10
- Rarely: 15
- Never: 0
## Software Metrics in Scientific Software: Knowledge and Use of Metrics

<table>
<thead>
<tr>
<th>Category</th>
<th>Number of Unique Metrics</th>
<th>Known (frequency)</th>
<th>Used (frequency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Code Complexity</td>
<td>13</td>
<td>49</td>
<td>10</td>
</tr>
<tr>
<td>General Quality</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Methodology</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Performance</td>
<td>9</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>Process</td>
<td>9</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Recognition</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Testing</td>
<td>12</td>
<td>20</td>
<td>13</td>
</tr>
</tbody>
</table>
Code Review in Scientific Software

• Interviews and surveys of scientific software developers

• Goals
  • Understand code review process
  • Understand impacts and expectations
  • Understand barriers
  • Identify areas of potential improvement
Code Review in Scientific Software: Importance

- Large portion of code is reviewed
- Shared expertise improves code quality
- Consistent style and reusability
- Good for new contributors and tricky features
- Saves debugging time
Code Review in Scientific Software: Challenges

• Underlying science viewed as more important than code

• Developers are attached to the way they have done things and resist change

• Lack of time and qualified contributors

• Lack of enough people to properly review

• Obtaining reviewer agreement
Summary

• Scientific Software Engineering needs:
  • Diverse
  • Deep

• Unique problems that lack simple solutions

• Successful interactions require
  • Time
  • Openness to new ideas
Acknowledgements

- Roscoe Bartlett
- Victor Basili
- Neil Chue Hong
- Nasir Eisty – PhD student
- Thomas Epperly
- Christine Halverson
- Dustin Heaton – Former PhD student
- Lorin Hochstein
- Jeff Hollingsworth
- Dan Katz
- Richard Kendall
- Karla Morris
- Aziz Nanthaamornphong - Former PhD student
- Damian Rouson
- Forrest Shull
- Susan Squires
- Doug Post
- Marvin Zelkowicz
Further Readings: Community Surveys


Further Readings:
SE for CSE


Further Readings:
SE-CSE Workshops


Further Readings:
Case Studies


Further Readings:
Community Interactions