Methodology for Assessing the State of the Practice for Domain X

Spencer Smith
McMaster University, Canada
smiths@mcmaster.ca

Jacques Carette
McMaster University, Canada
carette@mcmaster.ca

Olu Owojaiye
McMaster University, Canada
owojaiyo@mcmaster.ca

Peter Michalski
McMaster University, Canada
michap@mcmaster.ca

Ao Dong
McMaster University, Canada
donga9@mcmaster.ca

Abstract

To improve software development methods and tools for research software, we first need to understand the current state of the practice. Therefore, we have developed a methodology for assessing the state of the software development practices for a given research software domain. The methodology is applied to one domain at a time in recognition that software development in different domains is likely to have adopted different best practices. Moreover, providing a means to measure different domains facilitates comparison of development practices between domains. For each domain we wish to answer questions such as: i) What artifacts (documents, code, test cases, etc.) are present? ii) What tools are used? iii) What principles, process and methodologies are used? iv) What are the pain points for developers? v) What actions are used to improve qualities like maintainability and reproducibility? To answer these questions, our methodology prescribes the following steps: i) Identify the domain; ii) Identify a list of candidate software packages; iii) Filter the list to a length of about 30 packages; iv) Gather source code and documentation for each package; v) Collect repository related data on each software package, like number of stars, number of open issues, number of lines of code; vi) Fill in the measurement template (the template consists of 108 questions to assess 9 qualities (including the qualities of installability, usability and visibility)); vii) Interview developers (the interview consists of 20 questions and takes about an hour); viii) Rank the software using the Analytic Hierarchy Process (AHP); and, ix) Analyze the data to answer the questions posed above. A domain expert should be engaged throughout the process, to ensure that implicit information about the domain is properly represented and to assist with conducing an analysis of the commonalities and variabilities between the 30 selected packages. Using our methodology, spreadsheet templates and AHP tool, we estimate (based on our experience with using the process) the time to complete an assessment for a given domain at 173 person hours.

2012 ACM Subject Classification Software and its engineering; Software and its engineering → Software product lines; General and reference → Empirical studies

Keywords and phrases software quality, domain analysis, scientific computing software, research software, empirical measures, analytic hierarchy process

Contents

1 Introduction

Research software uses computing to simulate mathematical models of real world systems so that we can better understand and predict those systems' behaviour. A small set of examples of important research software includes the following: designing new automotive parts, analyzing the flow of blood in the body, and determining the concentration of a pollutant released into the groundwater. As these examples illustrate, research software can be used for tackling important problems that impact such areas as manufacturing, financial planning, environmental policy, and the health, welfare and safety of communities.

Given the importance of research software, scientists and engineers are pushing for methods and tools to sustainably develop high quality software. This is evident from the existence of such groups as the Software Sustainability Institute (SEI) and Better Scientific Software (BSS). Sustainability promoting groups such as these are necessary because unfortunately the current "state of the practice" for research software often does not incorporate "state of the art" Software Engineering (SE) tools and methods (Johanson and Hasselbring, 2018). The lack of SE tools and methods contributes to sustainability and reliability problems (Faulk et al., 2009). Problems with the current state of the practice are evident from embarrassing failures, like a retraction of derived molecular protein structures (Miller, 2006), false reproduction of sonoluminescent fusion (Post and Votta, 2005), and fixing and then reintroducing the same error in a large code base three times in 20 year (Milewicz and Raybourn, 2018). To improve this situation, we need to first fully understand the current state of the practice for research software development.

The purpose of our proposed methodology is to understand how software quality is impacted by the software development principles, processes and tools currently used within research software communities. Since research software is so broad a category, we will reduce the scope of our methodology to focus on one specific research software domain at a time. To emphasize that the method is generic, we will label the specific domain as X within this document.

This "state of the practice for domain X" exercise builds off of prior work on measuring/assessing the state of software development practice in several research domains. We have updated the work that was done previously for domains such as Geographic Information Systems (Smith et al., 2018a), Mesh Generators (Smith et al., 2016a), Seismology software (Smith et al., 2018c), and Statistical software for psychology (Smith et al., 2018b). Initial tests of the new methodology have been done for medical image analysis software (Dong, 2021) and for Lattice Boltzmann Method (LBM) software (Michalski, 2021).

In the previous "state of the practice" project, we measured 30 software projects for each domain, but the measures were relatively shallow. With this re-boot we still target about 30 software examples from each domain, but we are now collecting more data. In keeping with the previous project, we still have the constraint that the work load for applying the methodology to a given domain needs to be feasible for a team as small as one individual, and for a time that is short, ideally around a person month per domain.[1]

To begin our re-boot of the previous methodology we critically assessed, and subsequently modified, our previous set of measures. In addition, the following data has been added to the new methodology:

- Characterization of the functionality provided by the software in the domain via a

[1] A person month is considered to be 20 working days (4 weeks in a month, with 5 days of work per week) at 8 person hours per day, or $20 \cdot 8 = 160$ person hours.

commonality analysis.
- Project repository related data, such as the number of files, number of lines of code, percentage of issues that are closed, etc.
- Interviews with software developers in domain X.

Unlike the previous measurement process, the new methodology involves and engages a domain expert partner throughout. We did not previously engage the domain expert with the rationale that we wished to eliminate potential bias. However, this advantage is not worth the inability to evaluate the functionality of the software. Moreover, not having an expert makes navigating the on-line resources difficult, since on-line resources are often silent on information that is implicit to domain experts. Furthermore, not every statement found on-line will necessarily be accurate. The importance of the domain expert is particularly noteworthy when it comes time for publication and dissemination of the state of the practice assessment. Throughout this document the person (or persons) that provide domain expertise will be designated as the *Domain Expert*.

In the proposed methodology, the collected data is combined to rank the software within each domain using the Analytic Hierarchy Process (AHP) (Saaty, 1980). As in the previous measurement exercise, we use AHP to develop a list of software ranked by quality. However, in the new process we do not stop with this list. The Domain Expert is consulted to verify the ordering, and to discuss the decisions that led to the ranking. The AHP process is used to facilitate a conversation with the Domain Expert as a means to deepen our understanding of the software in the domain, and the needs of typical developers.

So that the collected data for a given domain can benefit the scientific community, our recommendation is that all collected data be made public. For instance, the data collection for each domain can be put on a GitHub repository. In addition to the project record left on GitHub, the final data can be exported to Mendeley Data. As an example, the measurements for the state of the practice for GIS software are available on Mendeley. Ideally the full analysis of the state of the practice for domain X will also be published in a suitable journal, allowing for dissemination/feedback and communication.

The scope of this methodology includes observations on product, artifact (documentation, test scripts, etc.) and process quality for research software. We leave the assessment of the performance of research software, for instance using benchmarks, to other projects, such as the work of Kågström et al. (1998). Currently we are also leaving experiments to measure usability and modifiability as future work, as discussed in Section 13.

Within this document the following typographic conventions have been adopted: i) Red text denotes a link to sections of this document. ii) Cyan text denotes a URL link. iii) Blue text denotes a citation.

The full methodology is presented in the sections that follow. Section 2 highlights the research questions that are to be answered for each measured domain. These questions are answered by the data collected using the process outlined in Section 3. The major steps in the process are outlined in Sections 4 – 11. Following this, the time required for assessing a single domain is estimated in Section 12.

2 Research Questions

The following are the research questions that we wish to answer for each of our selected domains. In addition to answering the questions for each domain, we also wish to combine the data from multiple domains to answer these questions for research software in general.

1. What artifacts are present in current software packages?

2. What tools (development, dependencies, project management) are used by current software packages?
3. What principles, processes, and methodologies are used in the development of current software packages?
4. What are the pain points for developers working on research software projects? What aspects of the existing processes, methodologies and tools do they consider as potentially needing improvement? How should processes, methodologies and tools be changed to improve software development and software quality?
5. For research software developers, what specific actions are taken to address the following:
   a. usability
   b. traceability
   c. modifiability
   d. maintainability
   e. correctness
   f. understandability
   g. unambiguity
   h. reproducibility
   i. visibility/transparency
6. How does software designated as high quality by this methodology compare with top rated software by the community?

3 Overview of Steps in Assessing Quality of the Domain Software

To answer the above research questions (Section 2), we systematically measure the quality of the software through data collection. An overview of the measurement process is given in the following steps, starting from determining a domain that is suitable for measurement:

1. Identify the domain (X). (Section 4)
2. Ask the Domain Expert to create a top ten list of software packages in the domain. (Section 5)
3. Meet with the Domain Expert to brief them on the overall objective, research proposal, research questions, measurement template, and developer interviews. (Section 5)
4. Identify the broad list of candidate software packages in the domain. (Section 6)
5. Preliminary filter of software packages list. (Section 7)
6. Review software list with Domain Expert. (Section 5)
7. Domain Analysis (with the help of the Domain Expert). (Section 8)
8. Ask Domain Expert to vet domain analysis. (Section 5)
9. Gather source code and documentation for each prospective software package.
10. Collect repository based information. (Section 9)
11. Measure using measurement template. (Section 10)
12. Survey developers. (Section D)
13. Use AHP process to rank the software packages. (Section 11)
14. Ask Domain Expert to vet AHP ranking. (Section 5)
15. Answer research questions (from Section 2) and document answers.

4 How to Identify the Domain?

A domain of research software must be identified to begin the assessment. Research software is defined in this exercise as "software that is used to generate, process or analyze results that [are intended] to appear in a publication" (Hettrick et al., 2014) in a scientific or engineering context. Research software is a more general term for what is often called scientific computing. To be applicable for the methodology described in this document, the chosen domain must have the following properties:

1. The domain must have well-defined and stable theoretical underpinning. A good sign of this is the existence of standard textbooks, preferably understandable by an upper year undergraduate student.
2. There must be a community of people studying the domain.
3. The software packages must have open source options.
4. A preliminary search, or discussion with experts, suggests that there will be numerous, close to 30, candidate software packages in the domain that qualify as 'research software'.

Some examples of domains that fit these criteria are finite element analysis (Szabo and Actis, 1996), quantum chemistry (Veryazov et al., 2004), seismology (Smith et al., 2018c), as well as mesh generators (Smith et al., 2016b).

5 Interaction With Domain Expert

As mentioned in the introduction (Section 1), the Domain Expert is an important member of the state of the practice assessment team. Pitfalls exist if non-domain experts attempt to acquire an authoritative list of software, or perform a commonality analysis, or try to definitively rank the software. The main source of problems for non-domain experts is that they can only rely on information that is available on-line, but on-line data has two potential problems: i) the on-line resources could have false or inaccurate information; and, ii) the on-line resources could leave out relevant information that is so in-grained with experts that nobody thinks to explicitly record the information.

Domain experts may be recruited from academia or industry. The only requirements are knowledge of the domain and a willingness to be engaged in the assessment process. The Domain Expert does not have to be a software developer, but they should be a user of domain software. Given that the domain experts are likely to be busy people, the measurement process cannot put to much of a burden on their time.

In advance of the first meeting with the Domain Expert (Step 3 in Section 3) the expert is asked to create a top ten list of software packages in the domain. This is done to help the expert get in the right mind set in advance of the first meeting. Moreover, by doing the exercise in advance, we avoid the potential pitfall of the expert approving the discovered list of software without giving it adequate thought. The emphasis during the first meeting is for the Domain Expert to learn what is expected of them. The discussion should also cover avenues for publication and dissemination.

The Domain Experts are asked to vet the collected data and analysis. In particular, they are asked to vet the proposed list of software packages, the domain analysis and the AHP ranking. These interactions can be done either electronically or with in-person (or virtual) meetings.

6 How to Identify Candidate Software?

Once the domain of interest is identified, the candidate software for measuring can be found through search engine queries targeting authoritative lists of software. Potential places to search include GitHub, swMATH and domain related publications, such as review articles. Domain Experts are also asked for their suggestions and are asked to review the initial draft of the software list.

When forming the list and reviewing the candidate software the following properties should be considered:

1. The software functionality must fall within the identified domain.
2. The source code must be viewable.
3. The empirical measures listed in Section 9 should ideally be available, which implies a preference for GitHub-style repositories.
4. The software cannot be marked as incomplete, or in an initial development phase.

7 How to Initially Filter the Software List?

If the list of software is too long (over around 30 packages), then steps need to be taken to create a more manageable list. To reduce the length of the list, the following filters are applied. The filters are applied in the priority order listed, with the filtering process stopped once the list size is manageable.

1. Scope: Software is removed by narrowing what functionality is considered to be within the scope of the domain.
2. Usage: Software packages are eliminated if their installation procedure is not clear and easy to follow.
3. Age: The older software packages (age being measured by the last date when a change was made) are eliminated, except in the cases where an older software package appears to be highly recommended and currently in use. (The Domain Expert can be consulted on this question, if necessary.)

Copies of both the initial and filtered lists, along with the rationale for shortening the list, should be kept for traceability purposes.

8 Domain Analysis

Since each domain we will study will have a reasonably small scope, we will be able to view the software as constituting a program family. The concept of a program family is defined by Parnas (1976) as “a set of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members”. Studying the common properties within a family of related programs is termed a domain analysis.

The domain analysis consists of a commonality analysis of the family of software packages. Its purpose is to show the relationships between these packages, and to facilitate an understanding of the informal specification and development of them. Weiss (1997) defines commonality analysis as an approach to defining a family by identifying commonalities, variabilities, and common terminology for the family. Commonalities are goals, theories, models, definitions and assumptions that are common between family members. Variabilities are goals, theories, models, definitions and assumptions that differ between family members.

Associated with each variability are its parameters of variation, which summarize the possible values for that variability, along with their potential binding time. The binding time is when the value of the variability is set. It could be set as specification time, build time (when the program is being compiled) or run time (when the code is executing).

The final result of the domain analysis will be tables of commonalities, variabilities, and parameters of variation of a program family. Smith et al. (2008) present a template for conducting a commonality analysis, which was referred to when conducting this work. Weiss (1998) describes another commonality analysis technique for deciding the members of a program family. Smith and Chen (2004) and Smith et al. (2017) are examples of a commonality analysis for a family of mesh generating software and a family of material models, respectively. The steps to produce a commonality analysis are:

1. Write an Introduction
2. Write an Overview of the Domain Knowledge
3. List Commonalities
4. List Variabilities
5. List Parameters of Variation
6. Add Terminology, Definitions, Acronyms

9 Repository Based Measures

Some quality measurements rely on gathering raw and processed data from software repositories. We focus on data that is reasonably easy to collect, which we combine and analyze. The measures that are collected relate to the research questions (Section 2). For instance, we collect data to see how large a project is, to ascertain a project's popularity, and to determine whether to project is being actively developed.

Section 9.1 lists the raw data that is collected. Some of this data can be observed from GitHub repository metrics. The rest can be collected using freeware tools. GitStats is used to measure the number of binary files as well as the number of added and deleted lines in a repository. This tool is also used to measure the number of commits over different intervals of time. Sloc Cloc and Code (scc) is used to measure the number of text based files as well as the number of total, code, comment, and blank lines in a repository. These tools were selected due to their installability, usability, and ability to gather the empirical measures listed below. Details on installing and running the tools can be found in Appendix A. Section 9.2 introduces the required processed data, which is calculated using the raw data.

9.1 Raw Data

The following raw data measures are extracted from repositories:

- Number of stars.
- Number of forks.
- Number of people watching the repository.
- Number of open pull requests.
- Number of closed pull requests.
- Number of developers.
- Number of open issues.
- Number of closed issues.
- Initial release date.

- Last commit date.
- Programming languages used.
- Number of text-based files.
- Number of total lines in text-based files.
- Number of code lines in text-based files.
- Number of comment lines in text-based files.
- Number of blank lines in text-based files.
- Number of binary files.
- Number of total lines added to text-based files.
- Number of total lines deleted from text-based files.
- Number of total commits.
- Numbers of commits by year in the last 5 years. (Count from as early as possible if the project is younger than 5 years.)
- Numbers of commits by month in the last 12 months.

9.2 Processed Data

The following measures are calculated from the raw data:

- Status of software package as either dead or alive, where alive is defined as the presence of repository commits or software package version releases in the last 18 months.
- Percentage of identified issues that are closed.
- Percentage of code that is comments.

The time frame of 18 months was selected as the separating point between alive and dead projects because this is the usual timeframe for operating system updates.

10 Measure Using Measurement Template

The Measurement Template is found in Appendix B. This template is used to track measurements and quality scores for all of the software packages in the domain. For each software package, we fill-in the template questions. This process can take between 1 to 4 hours for each package. Project developers can be contacted for help regarding installation, if necessary, but a cap of about 2 hours should be imposed on the entire installation process, to keep the overall measurement time feasible. To save time, a blank measurement template spreadsheet has been prepared, with the measures as rows. An excerpt of the spreadsheet is shown in Figure 1. A column should be added to this template for each software package to be measured.

The full template consists of 108 questions categorized under 9 qualities. The questions were designed to be unambiguous, quantifiable and measurable with limited time and domain knowledge. The measures are grouped under headings for each quality, and one for summary information. The summary information (shown in Figure 1) is the first section of the template. This section summarizes general information, such as the software name, number of developers, etc. We follow the definitions given by Gewaltig and Cannon (2012) for the software categories. Public means software intended for public use. Private means software aimed only at a specific group, while the concept category is used for software written simply to demonstrate algorithms or concepts. The three categories of development models are: open source, where source code is freely available under an open source license; free-ware, where a binary or executable is provided for free; and, commercial, where the user must pay for the software product.

	A	B	C	D
1	**Metrics & Description**	**Possible Measurement Values**		
2	**Summary Information**	**⋆ is used to indicate that a response of this type should be accompanied by explanatory text**	**{software package 1}**	**{software package 2}**
3	Software name?	(string)		
4	URL?	(URL)		
5	Affiliation (institution(s))	(string or {N/A})		
6	Software purpose	(string)		
7	Number of developers (all developers that have contributed at least one commit to the project) (use repo commit logs)	(number)		
8	How is the project funded?	(unfunded, unclear, funded*) where * requires a string to say the source of funding		
9	Initial release date?	(date)		
10	Last commit date?	(date)		
11	Status? (alive is defined as presence of commits in the last 18 months)	({alive, dead, unclear})		
12	License?	({GNU GPL, BSD, MIT, terms of use, trial, none, unclear, other*}) * given via a string		
13	Platforms?	(set of {Windows, Linux, OS X, Android, other*}) * given via string		
14	Software Category? The concept category includes software that does not have an officially released version. Public software has a released version in the public domain. Private software has a released version available to authorized users only.	({concept, public, private})		
15	Development model?	({open source, freeware, commercial, unclear})		
16	Publications about the software? Refers to publications that have used or mentioned the software	(number or {unknown})		

Sheet1

Figure 1 Excerpt of the Top Section of the Measurement Template (Summary Information)

Following the summary section are sections to measure 9 qualities: **1.** installability; **2.** correctness and verifiability; **3.** surface reliability; **4.** surface robustness; **5.** surface usability; **6.** maintainability; **7.** reusability; **8.** surface understandability; and, **9.** visibility/transparency. Definitions of these qualities are available in a working document on software quality. Several of the qualities use the word "surface". This is to highlight that, for these qualities in particular, the best that we can do is a shallow measure of the quality. For instance, we are not currently doing any experiments to measure usability. Instead, we are looking for an indication that usability was considered by the developers. We do this by looking for cues in the documentation, like a getting started manual, a user manual and documentation of expected user characteristics.

Most of the data to be collected should be straightforward from reviewing the measurement template. However, in a few cases extra guidance is necessary to eliminate ambiguity, as follows:

1. Initial release date: Mark the release year if an exact date is not available.
2. Publications about the software: A list of publications can be found directly on the website of some software packages. For others use Google Scholar or a similar index.
3. Is there evidence that performance was considered?: Search the software artifacts for any mention of speed, storage, throughput, performance optimization, parallelism, multi-core processing, or similar considerations. The search function on GitHub can help.
4. Getting started tutorial: Sometimes this is found within another artifact, like the user manual.
5. Continuous integration: Search the software artifacts for any mention of continuous integration. The search function on GitHub can help. In some cases, `yaml` files will provide a hint that continuous integration is employed.

To fill-in the spreadsheet template, the following steps should be followed:

1. Gather the summary information into the top section of the document (Figure 1).

2. Using the GitStats tool that is described in Section 9 gather the measurements for the Repo Metrics - GitStats section found near the bottom of the spreadsheet.
3. Using the SCC tool that is also described in Section 9 gather the measurements for the Repo Metrics – SCC section found near the bottom of the spreadsheet.
4. If the software package is found on git, gather the measurements for the Repo Metrics - the GitHub section found near the bottom of the spreadsheet.
5. Review installation documentation and attempt to install the software package on a virtual machine.
6. Gather the measurements for installability
7. Gather the measurements for correctness and verifiability
8. Gather the measurements for surface reliability
9. Gather the measurements for surface robustness
10. Gather the measurements for surface usability
11. Gather the measurements for maintainability
12. Gather the measurements for reusability
13. Gather the measurements for surface understandability
14. Gather the measurements for visibility and transparency
15. Assign a score out of ten for each quality. The score can be measured using the Measurement Template Impression Calculator, found in Appendix C. For each quality measurement, the file indicates the appropriate score to assign the measurement based on possible measurement values.

As in Smith et al. (2016a), Virtual machines (VMs) are used to provide an optimal testing environments for each package. VMs were used because it is easier to start with a fresh environment without having to worry about existing libraries and conflicts. Moreover, when the tests are complete the VM can be deleted, without any impact on the host operating system. The most significant advantage of using VMs is to level the playing field. Every software install starts from a clean slate, which removes “works-on-my-computer“ errors. When filling in the measurement template spreadsheet, the the details for each VM should be noted, including hypervisor and operating system version.

11 Analytic Hierarchy Process

The Analytical Hierarchy Process (AHP) is a decision-making technique that can be used when comparing multiple options by multiple criteria. In our work AHP is used for comparing and ranking the software packages of a domain using the quality scores that are gathered in the Measurement Template (Appendix B). AHP performs a pairwise analysis between each of the 9 quality options for each of the 30 software packages. This results in a matrix, which is used to generate an overall score for each software package for the given criteria. Smith et al. (2016a) shows how AHP is applied to ranking software based on quality measures. We have developed a tool for conducting this process. The tool includes an AHP JAR script and a sensitivity analysis JAR script that is used to ensure that the software package rankings are appropriate with respect to the uncertainty of the quality scores. The README file outlines the requirements for, and configuration and usage of, the JAR scripts. The JAR scripts, source code, and required libraries are located in the same folder as the README file.

12 Estimate of Time Required

Table 1 estimates the time required (in person hours) to complete a state of the practice assessment for domain X. The table assumes that the domain has already been decided and the Domain Expert has been recruited. The time spent by the Domain Expert is not included in the numbers shown in the table, since the amount of time that the domain expert will work independently of the rest of the assessment team will be small. Moreover, this amount of time will vary greatly depending on the preferred work habits of the Domain Expert. The table follows the steps outlined in Section 3. Time is not included for reviewing the methodology. Moreover, it is assumed that the template spreadsheets linked in this document, and the developed AHP tool, will be employed, rather than developing new tools. The person hours given are a rough estimate, based on our experience completing assessments for medical image analysis software (Dong, 2021) and for Lattice Boltzmann Method (LBM) software (Michalski, 2021). These two domains were assessed at the same time as designing the methodology presented in this document. We did our best to estimate the time spent on measurement and separate it from the time spend on design and development. The estimate assumes 30 software packages will be measured; the numbers will need to be adjusted if the total packages changes.

Table 1 Estimated Person Hours for Assessing the State of Practice for Domain X

Task	Hours
Initial 1 hour meeting with the Domain Expert plus meeting prep	5
Identify broad list of candidate software (Section 6)	12
Filter software list (Section 7) (10 minutes per package for 30 packages)	5
Review software list with Domain Expert (Section 5)	2
Domain analysis (with help of Domain Expert) (Section 8)	20
Vet domain analysis with Domain Expert (Section 5)	3
Gather source code and documentation for each package (10 minutes per package for 30 packages))	5
Collect repository based data (Section 9) (10 minutes per package for 30 packages)	5
Measure using measurement template (Section 10) (2.5 hours per repo for 30 repos)	75
Solicit developers for interviews	2
Conduct interviews (1.5 hour interviews with 10 developers (assuming 1 in 3 developers agree to an interview))	15
AHP ranking	2
Work with Domain Expert to vet AHP ranking	2
Analyze data and answer research questions	20
Total	**173**

The total number of person hours is 173 hours. This is close to our goal of 1 month of person hours (160 hours). The amount of time spent by the Domain Expert can be estimated by summing the Domain Expert items in Table 1 and adding an estimate of the time that they will independently spend on their assigned tasks. If we assume that the Domain Expert will spend 2 hours on the domain analysis and another 2 hours with answering questions, the Domain Expert time will be about 12 person hours.

13 Future Work

- usability and modifiability experiments

Usability experiments (Experiments)

Modifiability experiments (Experiments)

- not just for research software

- use it to inform research on Drasil - what artifacts, what process dev currently use, how could the new process promoted by Drasil fit into the current development processes?

- meta analysis of collected data. Already have GIS, mesh generator. Currently working on medical image analysis and LBM.

14 Concluding Remarks

- summary

A A Guide to Repository-Based Measurement Tools

This appendix covers the tools used for collecting repository based data (Section 9). The two tools covered are git_stats and scc. The tools do not have to be used in any particular order.

A.1 git_stats

A.1.1 Introduction

Source Code: GitHub repo

A.1.2 User Manual

Official Manual: GitHub repo

A.1.3 Demo of Installation and Running the Tool

The installation steps on your machine may be different from this section. Please refer to the user manual mentioned in Section A.1.2, if necessary. The steps shown here were executed on a virtual machine with 8 cores and 16 GB RAM running Debian GNU/Linux 9.11.

1. Install ruby/gem environment

   ```
   apt-get install ruby ruby-nokogiri ruby-nokogiri-diff ruby-nokogumbo
   ```

   Check the installation:

   ```
   gem --version
   ```

2. Install the tool

   ```
   sudo gem install git_stats
   ```

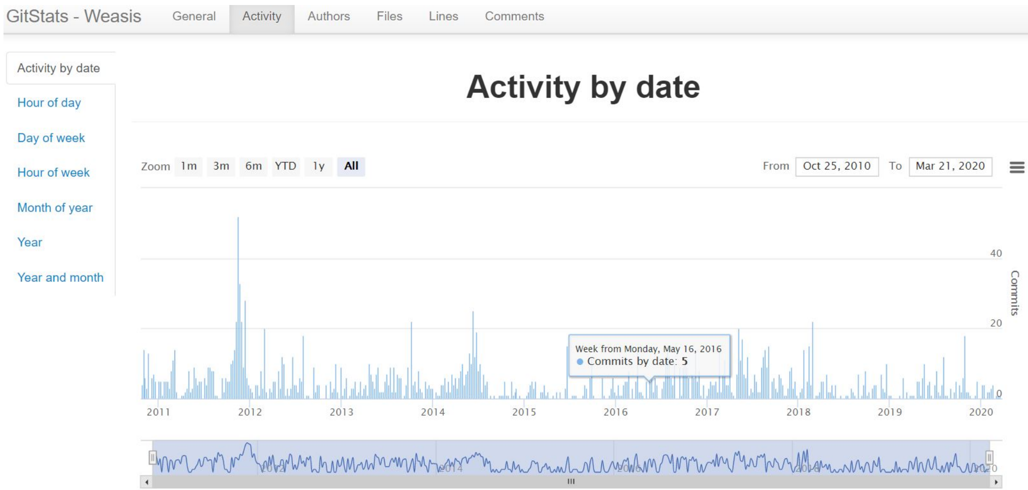
3. Prepare the target repo
   Make sure the target repo (the repo to be analyzed, not the repo of this tool) is on your machine. In this demo, the target repo is downloaded from a GitHub repo:

   ```
   # change [git path] to the url of your target repo
   git clone [git path]
   # e.g. git clone https://github.com/nroduit/Weasis.git
   ```

4. Generate analytics

   ```
   # make sure [repo path] is the target repo path
   # the [output path] can be anywhere you desire
   git_stats generate -p [repo path] -o [output path]
   # e.g. git_stats generate -p /home/user/git-stats/Weasis -o
   # /home/user/git-stats/Weasis-analytics
   ```

5. View the analytics
   View the analytic results by open [output path]/index.html with any browser or other software supporting HTML web page format.
6. Download the data
   On most of the taps of this web page, the data can be downloaded for more analytics by clicking the menu button beside the data-range section.



A.2 scc

A.2.1 Introduction

Source Code: GitHub repo

A.2.2 User Manual

Official Manual: GitHub repo

A.2.3 Demo of Installation and Running the Tool

The installation steps on your machine may be different from this section. Please refer to the user manual mentioned in Section A.2.2, if necessary. These steps were executed on a virtual machine with 8 cores and 16 GB RAM running Debian GNU/Linux 9.11

1. Install Golang
   Follow the official instructions, or the following demo,
   download the installation package:

   ```
   wget https://dl.google.com/go/go1.14.3.linux-amd64.tar.gz
   ```

   unpack to /usr/local:

   ```
   sudo tar -C /usr/local -xzf go1.14.3.linux-amd64.tar.gz
   ```

   use a text editor to open ~/.profile, e.g.:

   ```
   nano ~/.profile
   ```

   add the following lines to the end of this file:

   ```
   export GOPATH=$HOME/go
   export PATH=$PATH:/usr/local/go/bin:$GOPATH/bin
   ```

   save the file, and load the commands into the current shell instance:

   ```
   source ~/.profile
   ```

check the installation:

```
go version
```

2. Install the tool

```
go get -u github.com/boyter/scc/
```

3. Prepare the target repo
Make sure the target repo (the repo to be analyzed, not the repo of this tool) is on your machine. In this demo, the target repo is downloaded from a GitHub repo:

```
# change [project path] to your desired folder
cd [project path]
git clone https://github.com/nroduit/Weasis.git
```

4. Generate analytics

```
# make sure [repo path] is the target repo path
cd [repo path]
# use scc to generate analytics
scc
```

5. View the analytics
The results will be appear something like the following.

```
Language                 Files     Lines    Blanks  Comments      Code Complexity

Java                       745    129067     17549     13709     97809      18207
Properties File            119      7632       411      1116      6105          0
XML                        113      9291       163       272      8856          0
Plain Text                  44     10778        50         0     10728          0
Shell                        7       609        77       112       420         43
Markdown                     6      2205       516         0      1689          0
XML Schema                   4       529        53         0       476          0
License                      2       288        51         0       237          0
gitignore                    2        45         8         8        29          0
YAML                         1        19         3         1        15          0

Total                     1043    160463     18881     15218    126364      18250

Estimated Cost to Develop $4,348,103
Estimated Schedule Effort 26.819132 months
Estimated People Required 19.204800
```

B Measurement Template

The table below lists the full set of measures that are assessed for each software product. The measures are grouped under headings for each quality, and one for summary information. Following each measure, the type for a valid result is given in brackets. Many of the types are given as enumerated sets. For instance, the response on many of the questions is one of "yes," "no," or "unclear." The type "number" means natural number, a positive integer. The types for date and url are not explicitly defined, but they are what one would expect from their names. In some cases the response for a given question is not necessarily limited to one answer, such as the question on what platforms are supported by the software product. Case like this are indicated by "set of" preceding the type of an individual answer. The type in these cases are then the power set of the individual response type. In some cases a superscript * is used to indicate that a response of this type should be accompanied by explanatory text. For instance, if problems were caused by uninstall, the reviewer should note what problems were caused. The template also include 3 sections at the end for summarizing the repository based metrics. A blank measurement template spreadsheet is available to save time with data entry.

Table 2 Measurement Template

Summary Information
Software name? (string)
URL? (URL)
Affiliation (institution(s)) (string or N/A)
Software purpose (string)
Number of developers (all developers that have contributed at least one commit to the project) (use repo commit logs) (number)
How is the project funded? (unfunded, unclear, funded*) where * requires a string to say the source of funding
Initial release date? (date)
Last commit date? (date)
Status? (alive is defined as presence of commits in the last 18 months) (alive, dead, unclear)
License? (GNU GPL, BSD, MIT, terms of use, trial, none, unclear, other*) * given via a string
Platforms? (set of Windows, Linux, OS X, Android, other*) * given via string
Software Category? The concept category includes software that does not have an officially released version. Public software has a released version in the public domain. Private software has a released version available to authorized users only. (concept, public, private)
Development model? (open source, freeware, commercial, unclear)
Publications about the software? Refers to publications that have used or mentioned the software. (number or unknown)
Source code URL? (set of url, n/a, unclear)
Programming language(s)? (set of FORTRAN, Matlab, C, C++, Java, R, Ruby, Python, Cython, BASIC, Pascal, IDL, unclear, other*) * given via string
Is there evidence that performance was considered? Performance refers to either speed, storage, or throughput. (yes*, no)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Installability (Measured via installation on a virtual machine.)
Are there installation instructions? (yes, no)
Are the installation instructions in one place? Place referring to a single document or web-page. (yes, no, n/a)
Are the installation instructions linear? Linear meaning progressing in a single series of steps. (yes, no, n/a)
Are the instructions written as if the person doing the installation has none of the dependent packages installed? (yes, no, unclear)
Are compatible operating system versions listed? (yes, no)
Is there something in place to automate the installation (makefile, script, installer, etc)? (yes*, no)
If the software installation broke, was a descriptive error message displayed? (yes, no, n/a)
Is there a specified way to validate the installation? (yes*, no)
How many steps were involved in the installation? (Includes manual steps like unzipping files) Specify OS. (number, OS)
What OS was used for the installation? (Windows, Linux, OS X, Android, other*) *given via string
How many extra software packages need to be installed before or during installation? (number)
Are required package versions listed? (yes, no, n/a)
Are there instructions for the installation of required packages / dependencies? (yes, no, n/a)
Run uninstall, if available. Were any obvious problems caused? (yes* , no, unavail)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Correctness and Verifiability
Any reference to the requirements specifications of the program or theory manuals? (yes* , no, unclear)
What tools or techniques are used to build confidence of correctness? (literate programming, automated testing, symbolic execution, model checking, assertions used in the code, Sphinx, Doxygen, Javadoc, confluence, unclear, other*) * given via string
If there is a getting started tutorial? (yes, no)
Are the tutorial instructions linear? (yes, no, n/a)
Does the getting started tutorial provide an expected output? (yes, no*, n/a)
Does your tutorial output match the expected output? (yes, no, n/a)
Are unit tests available? (yes, no, unclear)
Is there evidence of continuous integration? (for example mentioned in documentation, Jenkins, Travis CI, Bamboo, other) (yes*, no, unclear)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Surface Reliability
Did the software "break" during installation? (yes* , no)
If the software installation broke, was the installation instance recoverable? (yes, no, n/a)
Did the software "break" during the initial tutorial testing? (yes*, no, n/a)
If the tutorial testing broke, was a descriptive error message displayed? (yes, no, n/a)
If the tutorial testing broke, was the tutorial testing instance recoverable? (yes, no, n/a)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Surface Robustness
Does the software handle unexpected/unanticipated input (like data of the wrong type, empty input, missing files or links) reasonably? (a reasonable response can include an appropriate error message.) (yes, no*)
For any plain text input files, if all new lines are replaced with new lines and carriage returns, will the software handle this gracefully? (yes, no*, n/a)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Surface Usability
Is there a getting started tutorial? (yes, no)
Is there a user manual? (yes, no)
Are expected user characteristics documented? (yes, no)
What is the user support model? FAQ? User forum? E-mail address to direct questions? Etc. (string)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Maintainability
What is the current version number? (number)
Is there any information on how code is reviewed, or how to contribute? (yes*, no)
Are artifacts available? (List every type of file that is not a code file – for examples please look at the 'Artifact Name' column of https://gitlab.cas.mcmaster.ca/SEforSC/se4sc/-/blob/git-svn/GradStudents/Olu/ResearchProposal/Artifacts_MiningV3.xlsx) (yes*, no, unclear) *list via string
What issue tracking tool is employed? (set of Trac, JIRA, Redmine, e-mail, discussion board, sourceforge, google code, git, BitBucket, none, unclear, other*) * given via string
What is the percentage of identified issues that are closed? (percentage)
What percentage of code is comments? (percentage)
Which version control system is in use? (svn, cvs, git, github, unclear, other*) * given via string
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Reusability
How many code files are there? (number)
Is API documented? (yes, no, n/a)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)

Surface Understandability (Based on 10 random source files)
Consistent indentation and formatting style? (yes, no, n/a)
Explicit identification of a coding standard? (yes*, no, n/a)
Are the code identifiers consistent, distinctive, and meaningful? (yes, no* , n/a)
Are constants (other than 0 and 1) hard coded into the program? (yes, no* , n/a)
Comments are clear, indicate what is being done, not how? (yes, no* , n/a)
Parameters are in the same order for all functions? (yes, no* , n/a)
Is the name/URL of any algorithms used mentioned? (yes, no* , n/a)
Is code modularized? (yes, no* , n/a)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Visibility/Transparency
Is the development process defined? If yes, what process is used. (yes*, no, n/a)
Are there any documents recording the development process and status? (yes*, no))
Is the development environment documented? (yes*, no)
Are there release notes? (yes*, no)
Overall impression? (1 .. 10)
Additional comments? (can cover any metrics you feel are missing, or any other thoughts you have)
Raw Metrics (Measured via git_stats)
Number of text-based files. (number)
Number of binary files. (number)
Number of total lines in text-based files. (number)
Number of total lines added to text-based files. (number)
Number of total lines deleted from text-based files. (number)
Number of total commits. (number)
Numbers of commits by year in the last 5 years. (Count from as early as possible if the project is younger than 5 years.) (list of numbers)
Numbers of commits by month in the last 12 months. (list of numbers)

Raw Metrics (Measured via scc)
Number of text-based files. (number)
Number of total lines in text-based files. (number)
Number of code lines in text-based files. (number)
Number of comment lines in text-based files. (number)
Number of blank lines in text-based files. (number)
Repo Metrics (Measured via GitHub)
Number of stars. (number)
Number of forks. (number)
Number of people watching this repo. (number)
Number of open pull requests. (number)
Number of closed pull requests. (number)

C Measurement Template Impression Calculator

The table below lists how each quality measure of the measurement template is used to calculate an overall impression in each software quality set.

Table 3 Measurement Template

Installability (Measured via installation on a virtual machine.)
Are there installation instructions? (yes=1, no=-1)
Are the installation instructions in one place? Place referring to a single document or web-page. (yes=1, no=0, n/a=0)
Are the installation instructions linear? Linear meaning progressing in a single series of steps. (yes=1, no=0, n/a=0)
Are the instructions written as if the person doing the installation has none of the dependent packages installed? (yes=1, no=0, unclear=0)
Are compatible operating system versions listed? (yes=1, no=0)
Is there something in place to automate the installation (makefile, script, installer, etc)? (yes*=1, no=-1)
If the software installation broke, was a descriptive error message displayed? (yes=0, no=-2, n/a=1)
Is there a specified way to validate the installation? (yes*=1, no=0)
How many steps were involved in the installation? (Includes manual steps like unzipping files) Specify OS. (<10 = 1)
What OS was used for the installation? (does not count)
How many extra software packages need to be installed before or during installation? (<10 = 1)
Are required package versions listed? (yes=1, no=0, n/a=1)
Are there instructions for the installation of required packages / dependencies? (yes=1, no=0, n/a=1)
Run uninstall, if available. Were any obvious problems caused? (yes*=0, no=1, unavail=1)
Overall impression? (a sum of >10 is rounded down to 10)

Correctness and Verifiability
Any reference to the requirements specifications of the program or theory manuals? (yes*=2, no=0, unclear=0)
What tools or techniques are used to build confidence of correctness? (any=1, unclear=0)
If there is a getting started tutorial? (yes=2, no=0)
Are the tutorial instructions linear? (yes=1, no=0, n/a=0)
Does the getting started tutorial provide an expected output? (yes=1, no*=0, n/a=0)
Does your tutorial output match the expected output? (yes=1, no=0, n/a=0)
Are unit tests available? (yes=1, no=0, unclear=0)
Is there evidence of continuous integration? (for example mentioned in documentation, Jenkins, Travis CI, Bamboo, other) (yes*=1, no=0, unclear=0)
Surface Reliability
Did the software “break” during installation? (yes*=0, no=5)
If the software installation broke, was the installation instance recoverable? (yes=5, no=0, n/a=0)
Did the software “break” during the initial tutorial testing? (yes*=0, no=5, n/a=0)
If the tutorial testing broke, was a descriptive error message displayed? (yes=2, no=0, n/a=0)
If the tutorial testing broke, was the tutorial testing instance recoverable? (yes=3, no=0, n/a=0)
Surface Robustness
Does the software handle unexpected/unanticipated input (like data of the wrong type, empty input, missing files or links) reasonably? (a reasonable response can include an appropriate error message.) (yes=5, no*=0)
For any plain text input files, if all new lines are replaced with new lines and carriage returns, will the software handle this gracefully? (yes=5, no*=0, n/a=5)

Surface Usability
Is there a getting started tutorial? (yes=3, no=0)
Is there a user manual? (yes=4, no=0)
Are expected user characteristics documented? (yes=1, no=0)
What is the user support model? FAQ? User forum? E-mail address to direct questions? Etc. (one=1, two+=2, none=0)
Maintainability
What is the current version number? (provided=1, nothing=0)
Is there any information on how code is reviewed, or how to contribute? (yes*=1, no=0)
Are artifacts available? (List every type of file that is not a code file – for examples please look at the 'Artifact Name' column of https://gitlab.cas.mcmaster.ca/SEforSC/se4sc/-/blob/git-svn/GradStudents/Olu/ResearchProposal/Artifacts_MiningV3.xlsx) (Rate 0 – 2 depending on how many and perceived quality)
What issue tracking tool is employed? (nothing=0, email of other private=1, anything public or accessible by all devs (eg git) = 2)
What is the percentage of identified issues that are closed? (50%+=1, <50%=0)
What percentage of code is comments? (10%+=1, <10%=0)
Which version control system is in use? (anything=2, nothing=0)
Reusability
How many code files are there? (0-9=0, 10-49=1, 50-99=3, 100-299=4, 300-599=5, 600-999=6, 1000+=8)
Is API documented? (yes=2, no=0, n/a=0)
Surface Understandability (Based on 10 random source files)
Consistent indentation and formatting style? (yes=1, no=0, n/a=0)
Explicit identification of a coding standard? (yes*=1, no=0, n/a=0)
Are the code identifiers consistent, distinctive, and meaningful? (yes=2, no*=0, n/a=0)
Are constants (other than 0 and 1) hard coded into the program? (yes=1, no*=0, n/a=0)
Comments are clear, indicate what is being done, not how? (yes=2, no*=0, n/a=0)
Is the name/URL of any algorithms used mentioned? (yes=1, no*=0, n/a=0)
Parameters are in the same order for all functions? (yes=1, no*=0, n/a=0)
Is code modularized? (yes=1, no*=0, n/a=0)
Visibility/Transparency
Is the development process defined? If yes, what process is used. (yes*=3, no=0, n/a=0)
Are there any documents recording the development process and status? (yes*=3, no=0))
Is the development environment documented? (yes*=2, no=0)
Are there release notes? (yes*=2, no=0)

D Survey Questions for Developers

Introduction. Roadmap. Anything about traceability to research questions.
Information about these interview questions: This gives you an idea what I would like to learn about the development of domain software. Interviews will be one-to-one and will be open-ended (not just "yes or no" answers). Because of this, the exact wording may change a little. Sometimes I will use other short questions to make sure I understand what you told me or if I need more information when we are talking such as: "So, you are saying that ...?), to get more information ("Please tell me more?"), or to learn what you think or feel about something ("Why do you think that is...?").

D.1 Information about the developers and users

(Jegatheesan, 2016)

1. Interviewees' current position/title? degrees?
2. Interviewees' contribution to/relationship with the software?
3. Length of time the interviewee has been involved with this software?
4. How large is the development group?
5. Do you have a defined process for accepting new contributions into your team?
6. What is the typical background of a developer?
7. What is your estimated number of users? How did you come up with that estimate?
8. What is the typical background of a user?

D.2 Information about the software

Square brackets are used for traceability to qualities and relevant research questions found in Section ...

1. Currently, what are the most significant obstacles in your development process?
2. How might you change your development process to remove or reduce these obstacles?
3. How does documentation fit into your development process? Would improved documentation help with the obstacles you typically face? [traceability - research question 5b, visibility/transparency - research question 5i]
4. In the past, is there any major obstacle to your development process that has been solved? How did you solve it?
5. What is your software development model? For example, waterfall, agile, etc.
6. What is your project management process? Do you think improve this process can tackle the current problem? Were any project management tools used?
7. Was it hard to ensure the correctness of the software? If there were any obstacles, what methods have been considered or practiced to improve the situation? If practiced, did it work? [correctness - research question 5e]
8. When designing the software, did you consider the ease of future changes? For example, will it be hard to change the structure of the system, modules or code blocks? What measures have been taken to ensure the ease of future changes and maintains? [maintainability - research question 5d, modifiability - research question 5c]
9. Provide instances where users have misunderstood the software. What, if any, actions were taken to address understandability issues? [understandability - research question 5f]
10. What, if any, actions were taken to address usability issues? [usability - research question 5a]

11. Do you think the current documentation can clearly convey all necessary knowledge to the users? If yes, how did you successfully achieve it? If no, what improvements are needed? [unambiguity - research question 5g]

12. Do you have any concern that your computational results won't be reproducible in the future? Have you taken any steps to ensure reproducibility? [reproducibility - research question 5h]

References

Ao Dong. Assessing the state of the practice for medical imaging software. Master's thesis, McMaster University, Hamilton, ON, Canada, September 2021.

S. Faulk, E. Loh, M. L. V. D. Vanter, S. Squires, and L. G. Votta. Scientific computing's productivity gridlock: How software engineering can help. *Computing in Science Engineering*, 11(6):30–39, Nov 2009. ISSN 1521-9615. doi: 10.1109/MCSE.2009.205.

Marc-Oliver Gewaltig and Robert Cannon. Quality and sustainability of software tools in neuroscience. *Cornell University Library*, page 20 pp, May 2012.

Simon Hettrick, Mario Antonioletti, Leslie Carr, Neil Chue Hong, Stephen Crouch, David De Roure, Iain Emsley, Carole Goble, Alexander Hay, Devasena Inupakutika, et al. Uk research software survey 2014. 2014.

Thulasi Jegatheesan. Case studies in document driven design of scientific computing software. Master's thesis, McMaster University, Hamilton, Ontario, Canada, July 2016.

Arne N. Johanson and Wilhelm Hasselbring. Software engineering for computational science: Past, present, future. *Computing in Science & Engineering*, Accepted:1–31, 2018.

Bo Kågström, Per Ling, and Charles Van Loan. Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software (TOMS)*, 24(3):268–302, 1998.

Peter Michalski. State of the practice for lattice boltzmann method software. Master's thesis, McMaster University, Hamilton, Ontario, Canada, September 2021.

Reed Milewicz and Elaine M. Raybourn. Talk to me: A case study on coordinating expertise in large-scale scientific software projects. *ArXiv e-prints*, September 2018.

Greg Miller. SCIENTIFIC PUBLISHING: A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science*, 314(5807):1856–1857, 2006. doi: 10.1126/science.314.5807.1856. URL http://www.sciencemag.org.

David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, (1):1–9, 1976.

D. E. Post and L. G. Votta. Computational Science Demands a New Paradigm. *Physics Today*, 58(1):35–41, January 2005. doi: 10.1063/1.1881898.

T. L. Saaty. *The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation*. McGraw-Hill Publishing Company, New York, New York, 1980.

W. Spencer Smith and Chien-Hsien Chen. Commonality and requirements analysis for mesh generating software. In F. Maurer and G. Ruhe, editors, *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2004)*, pages 384–387, Banff, Alberta, 2004.

W Spencer Smith, Jacques Carette, and John McCutchan. Commonality analysis of families of physical models for use in scientific computing. In *Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering (SECSE08)*, 2008.

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. State of practice for mesh generation software. *Advances in Engineering Software*, 100:53–71, October 2016a.

W Spencer Smith, D Adam Lazzarato, and Jacques Carette. State of the practice for mesh generation and mesh processing software. *Advances in Engineering Software*, 100:53–71, 2016b.

W. Spencer Smith, John McCutchan, and Jacques Carette. Commonality analysis for a family of material models. Technical Report CAS-17-01-SS, McMaster University, Department of Computing and Software, 2017.

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. State of the practice for GIS software. https://arxiv.org/abs/1802.03422, February 2018a.

W. Spencer Smith, Yue Sun, and Jacques Carette. Statistical software for psychology: Comparing development practices between CRAN and other communities. https://arxiv.org/abs/1802.07362, 2018b. 33 pp.

W. Spencer Smith, Zheng Zeng, and Jacques Carette. Seismology software: State of the practice. *Journal of Seismology*, 22(3):755–788, May 2018c.

BA Szabo and RL Actis. Finite element analysis in professional practice. *Computer methods in applied mechanics and engineering*, 133(3-4):209–228, 1996.

Valera Veryazov, Per-Olof Widmark, Luis Serrano-Andrés, Roland Lindh, and Björn O Roos. 2molcas as a development platform for quantum chemistry software. *International journal of quantum chemistry*, 100(4):626–635, 2004.

David M Weiss. Defining families: The commonality analysis. *submitted to IEEE Transactions on Software Engineering*, 1997. URL http://www.research.avayalabs.com/user/weiss/Publications.html.

David M Weiss. Commonality analysis: A systematic process for defining families. In *International Workshop on Architectural Reasoning for Embedded Systems*, pages 214–222. Springer, 1998. URL citeseer.ist.psu.edu/13585.html.