

Improving Software Engineering in Academia

Samuel Grayson

Changes from last time

Of software virtues (e.g. correctness, adaptability, extensibility, readability, repeatability, maintainability), I have chosen to single out *correctness* and *adaptability*, because they are the easiest to motivate. Then, I separate those two as goals from *maintainability*, which is more like a method.

I have addressed going broad versus going deep, by saying I want to know what is broadly important, and then future work can deep-dive on individual practices (explained in Future Work). I have also changed “tools” to “classes of tools,” as individual tools go in and out of fashion.

I have written explicitly about threats to validity: self-assessment biases, biases in mined data, correlation doesn’t imply causation, and demographic confounding factors.

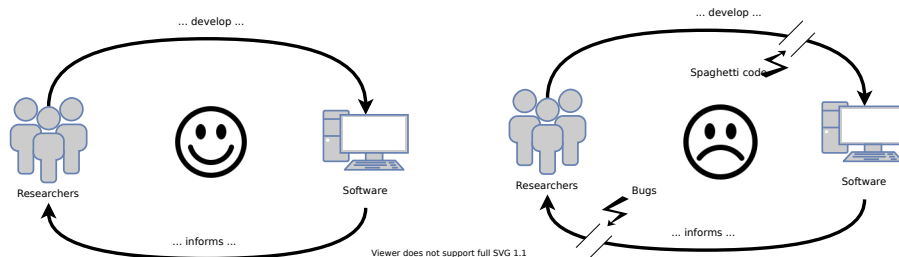
I have renamed “classes of software tools and management activities” to “software development practices (SDP).” I am open to new names, but they should be short or have acronyms.

I have included effort-estimation in the survey. This way I can answer which SDP has the greatest “bang for the buck” of effort.

Introduction¹

90% of researchers surveyed use *research software* to carry out their research, and more than half develop it themselves [4]. Researchers develop software, and software informs the researchers. However, incorrectness limits software’s ability to inform researchers, and non-adaptability makes it more difficult for researchers to develop software. Here, *correctness* means the the software behaves as the programmer expects it to, and *adaptability* means the software can be easily modified to support a change in functionality requirements or environment.

¹Darko and Dan, feel free to leave comments in the footnotes like this.



Researchers should care about *correctness* of software because software is a necessary condition for the correctness of the containing study despite being easy to get wrong. This makes it a ‘weak link’ for correctness. For example, Neupane et al. [5] find a bug in the code released by Willoughby et al. [8] five years after its publication, and by that time it had been cited more than 200 times. The bug was subtle enough to evade notice despite a high citation count, and the bug potentially calls into question the some of those 200 references.

Researchers should care about *adaptability* because small investments in adaptability will reduce the amount of time they themselves spend adapting their code to inevitable changes in requirements. Even “one-off” code still benefits from adaptability if it is developed iteratively.

Rosene defines *maintainability* as achieving the ability to trace causality backwards and being able to modify the program without breaking it [6]. Correctness and adaptability are properties of the behavior of software and not easy to measure directly; maintainability is a property of the software itself and possible to measure directly through quantitative metrics (TODO: state metrics and citations). But they are related: *maintainability implies adaptability* by its definition, and *maintainability implies a higher probability of correctness*, because it forces errors to be more noticeable (TODO: citations). For example, Hettrick originally wrote a data analysis in Excel formulae, but then rewrote them in scripts to improve maintainability and other software virtues [3]. In this process, Hettrick found two bugs in the original Excel implementation.

Research Questions

1. What is the distribution of maintainability in existing research software?
2. How effective per unit of effort is each *software development practice* (SDP; class of tools and activities) for ensuring maintainability? See the appendix for the SDP I want to test.

TODO: Write a logic model.

Prior work

Vandewalle determined the proportion of publications with available source code [7]. Availability important but distinct from maintainability, and maintainability is more immediately useful for the original authors. Furthermore, a common reason software is not available is that the authors are afraid it isn't maintainable enough [1], so improving maintainability might improve availability.

Eisty et al. have studied the effect of code review in great detail [2] and concluded that most research software engineers find the practice valuable. This is a good result, but it would be even better to be able to compare code review from other management activities. It might be the case that people will find all of them broadly useful, but they cannot implement every one of those practices due to resource constraints. Once this study identifies the most effective per unit of effort practices, one could study in depth those particular interventions.

TODO: more research on how to leverage learnings from software engineering in an industry-context. I suspect they will help, but that they might not transfer over exactly due to the open-ended nature of research software.

Methods

Survey

Next, I will seek out approval from administrators at institutions to email randomly-selected individual RSEs, PhD students, and post doctoral researchers. For RSEs, I can try University of Manchester RSDS, the University of Illinois NCSA, and University of Notre Dame CRC; for PhD students, I can try asking university departments where I have a connection. I plan to ask questions in the following categories:

- **SDP usage:** Which SDPs do you use?
- **Measures of maintainability:** Is your code maintainable in this specific way?
- **Confounding variables:** various other questions.
- **SDP value:** which SDP do you think is most valuable?

Pilot Survey

I plan to pilot this survey by asking people to complete it using a process identical to the real one. After they complete it, I will explain my research goal, and I will interview them afterwards. During this interview, I will ask closed- and open-ended questions like:

- Did you understand the questions?

- Do you think they are all relevant to maintainability, SDP use, confounding variables, or the research in general?
- Do you think that I missed a relevant question relevant to maintainability, SDP use, confounding variables, or the research in general?
- What is your reasoning for your answer to **SDP value**?

I plan to leverage my professional connections and those of my advisors to find people willing to talk to me about these subjects. I want to have five or ten of these thirty-minute interviews before continuing.

I will use the responses in the pilot survey to revise the survey.

Source-code Mining

I will analyze the source-code question from the survey to evaluate quantitative metrics in the following categories:

- **SDP usage**: Is this tool present in the source repository?
- **Measures of maintainability**: Evaluate quantitative metrics proposed by prior literature on software code quality.
- **Confounding variables**: Various quantitative metrics relating to the size and sort of software project.

Analysis

To answer RQ1, I will aggregate **measures of maintainability** in groups with **confounding variables** constant, from the survey and from data mining.

To answer RQ2, I will create a factor analysis from **SDP usage** to a D -dimensional space of latent “abstract maintainability” to **measures of maintainability**, separately for each group with near-enough **confounding variables**. Ideally, D will be 1 or else quite small. I can validate this by comparing my coefficients to the respondent’s estimate of **SDP value**. This analysis is has $(A+B)CD$ parameters to learn, where A is the number of SDP usage metrics, B is the number of measures of maintainability, and C is the number of equivalence classes of confounding variables. Contrast this with a linear model without factoring would have ABC parameters.

Finally, I will normalize value by **SDP cost** to find the “biggest bang for your buck.”

Threats to Validity

Content Validity

- The respondents in survey data could be biased in a particular direction, perhaps to over-estimate their software's maintainability and underestimate the time it takes to make changes. Even seasoned engineers have trouble estimating effort for changes.
 - However, the mined metrics can balance out bias in the survey.
- The metrics I have collected might not capture all details of maintainability.
 - However, I will read the literature and collect as much as I possibly can. Even partially capturing maintainability can still be useful, if that part is sufficient to capture the impact of tools on maintainability.
- The metrics might measure something other than maintainability.
 - As long as *most* of the metrics do correlate with maintainability, the statistical analysis will assign low coefficient to those metrics which are less related to maintainability.

Internal Validity

- We cannot establish causation solely from correlation.
 - However, use of SDP precedes the software and properties of the software. Future work can try an “intervention experiment” which can validate causality, once this study has narrowed down the candidates.
- Demographic factors could provide sources of confounding variance, particularly in self-assessments.
 - However, I can survey for basic demographic data (age, sex, experience), and control for that.
- The factor model might be too simple to capture the nuanced interplay of effects.
 - I can add a metric which is the product of two other metrics to model pair-wise interaction effects. If the main effects are significant by themselves, then they are still useful without considering all possible interaction effects.

External Validity

- There could be selection bias in those who choose to take part in the survey.

- This is my motivation for emailing individuals instead of posting the URL in a public place. This allows me to know how many people declined to take the survey, which will allow me to establish bounds for the generalizability of the study. I also plan to collect confounding variables, such as years of experience, which will help me know if the sample is skewed.
- An intervention that introduces a SDP is a different from that in which the developers chose that SDP. Perhaps the act of being told what to do reduces the efficacy of the practice.
 - Future work should do an intervention-study, to see if the results are generalizable.

TODO: Too few data

Conclusion

I hope to identify classes of tools and activities which will positively impact maintainability. Ideally, this will save time for the researchers who use them.

Future Work

In future work, one could execute an intervention experiment, where nascent software projects are randomly divided into groups. For one group, the experimenter could ask them to use certain classes of tools and activities, while the other receives no specific advice. If they adopt the tool or activity on their own accord, they would need to be shifted into the first group or ignored. Then, the experimenters could have more causal evidence of a tool improving software quality.

In future work, one could embed oneself in academic software engineering teams as Steve Easterbrook has (*I intend to write a citation here*).

In future work, one could examine automated metrics of software maintenance and survey human opinions of software maintenance. It would be useful if some metrics corresponded to human ground-truth. Then, one could look for tools and practices which maximize the automated metrics, which may be more scalable than asking in a survey if certain software is maintainable. However, one would need to be careful in writing recommendations, as these may become gamed rather than honestly sought ([Goodhart's Law](#)).

In future work, one could deep-dive on individual practices. For example, one might want to study details of code review: how many reviewers should review, what proportion of commits should require review, is it better to have junior or senior reviewers, etc.

Broader impact

1. The survey responses and data mining results may or may not indicate certain classes of tools or activities which correlate with maintainability.
2. I will investigate those classes of tools and activities more deeply, to learn how one should introduce them to new users.
3. I will evangelize the results.
4. Some researchers will change their behavior to include these tools or activities in their software development.
5. More research software will be correct and adaptable.

Item 1 is this work, which may possibly fail and end the chain immediately. 2 will be future work, possibly after I graduate. 3 is an long-term push for culture change by, for example, giving talks, that I hope will lead to 4 and 5.

Appendix

Software Development Practices

TODO: Define software tools and management activities. Software tools have the advantage of automating checks that would be otherwise time-consuming. Management activities require human input but still “automate” in some sense, since the carrying out the activity algorithmically necessarily draws attention to certain priorities.

Classes of Software Tools

- *Formatters* improve maintainability by automatically formatting your code, making it easier to read, and producing smaller diffs. Examples include black for Python, clang-fmt for C/C++.
- *Linters* improve maintainability by catching stylistic and surface-level “typo” bugs. Examples include pylint for Python, shellcheck for shell, clang-tidy for C/C++.
- *Static analyzers* improves maintainability by reasoning about user-code and identifying errors. Examples include mypy for Python, clang-analyze for C/C++.
- *Sanitizers* improve maintainability by running user-code with runtime-checks enabled. Examples include memory sanitizer, valgrind for C/C++.
- *Version control* improves maintainability by naming each stable version of code. It also enables continuous testing and feature branches. Examples include git, mercurial, and CVS.

- *Continuous testing* improves maintainability by running exhaustive tests and static analysis offline on each revision of the code. This can leverage static analyzers as well. Examples include Jenkins, GitLab CI, Travis CI, and GitHub Actions.
- *Test frameworks* improves maintainability by automating testing for regressions. This can be combined with continuous testing.
- *Inline documentation* improves maintainability explaining how the code works within the code. Some automated tools such as Javadoc, Doxygen, and Sphinx can extract and reformat inline documentation into a more readable form.
- *Workflow management tools* improve maintainability by making it easy to reuse intermediate results while having a fallback for fresh systems. Examples include Make, CMake, Bazel, and Popper.
- *Containerization tools* improve maintainability by specifying the software environment in which the code will run. Examples include Docker, Nix, and qemu, and even some workflow managers such as Bazel and Popper.

Classes of Management Activities

- *Test-driven development* improves maintainability by encouraging developers write tests before writing code. This increases the use of tests, which in turn make it easier to change software without breaking it.
- *Workflow hooks* improve maintainability by enforcing software tools at certain stages in the workflow, such as git pre-commit hooks.
- *Code review* improves maintainability by having the reviewer attempt to reproduce the result. It also improves maintainability by keeping the code quality high and spreading knowledge out across the team.
- *Agile development* and its variants seek to improve maintainability by constantly re-evaluating the software's requirements. TODO: Split Agile up into individual activities.
- *Feature branches* improves maintainability by separating a stable working state from experimentation. This can be used in combination with continuous testing.
- TODO: develop this section by adding more examples of activities that can improve maintainability.

Survey Questions

- **SDP usage:** For each project, for each SDP, did you use this SDP on this project?
- **Measure of maintainability:** For each project, how long in hours would it take you to run the code *with some possible perturbation* from raw data to publication-ready graphs, excluding compute time?
 - where the possible perturbation can be: no perturbation, a new dataset, a new machine, a new kind of plot, a new dimension of input data, etc.
- **SDP cost:** For each activity, how much of your time as a percentage does using tools or activities of this class require (increments of 10%)?
- **SDP value:** For each activity, how important is this SDP for maintainability (on a Likert scale)?
- **Confounding variable:** What is your job title (categorical with “other”)?
- **Confounding variable:** What is your field of research (categorical with “other”)?
- **Confounding variable:** How many years of experience do you have writing research software?
- **Source-code:** For each project, if you are comfortable doing so, please share the source code of this project.

Metrics Mined from Source-code

Measures of maintainability:

- For each language used, what is the cyclomatic complexity for code in that language?
- For each language used, what are the Halstead complexity measures?
- How much code near-duplicated code exists in the project?
- How many inline literals are used (aka “magic constants”)?
- How much coupling is there within the project?

SDP use:

- Is the repository repeatable (single executable that runs works out-of-the-box and exercises most of the code)?
 - Software projects often have bespoke systems for reproducing their project. To gather a large amount of data on reproducibility, I will write a program which can recognize patterns such as the existence of a `Dockerfile`, `Makefile`, or bazel BUILD file; whatever does not fit this pattern I will have to analyze manually. Hopefully, I can turn each exception into a new pattern that the program can handle next time.

- If so, what containerization/sandboxing tools does the repository employ?
- For each language used, what is the ratio of documentation to LoC in that language?
- How many items (e.g. methods, functions, classes) have structured documentation (aka “doc blocks”)?
- Are there automated tests?
- If so, what is their coverage?
- Does the repository have continuous testing?
- If so, through what tool?
- If so, what commands are being tested?
 - Classify by executables and subcommand (e.g. `docker build`, `cargo test`)

Confounding variable:

- Statement count
- SLoC count
- Programming language

TODO: Consider adding version-control history-based metrics, e.g. those from [PluralSight](#).

TODO: Consider adding community-oriented metrics, e.g. those from [CHAOS](#).

Time Table

Date	Task complete
Nov 15	Conduct informal interviews
Nov 15	Code and analyze informal interviews
Dec 15	Write survey questions
Dec 31	Apply for IRB approval
Dec 31	Ask administrators for approval
Jan 15	Identify software repositories to mine
Mar 1	Mine repositories
Feb 15	Mine citation counts
Mar 15	Mine other repositories for software usage
Mar 1	IRB approval
Mar 15	Deploy survey
Apr 1	Collect results of survey
Apr 15	Analyze data
May 1	Write manuscript

Feedback

- What works vs what doesn't
 - A logic model is a graphic depiction (road map) that presents the shared relationships among the resources, activities, outputs, outcomes, and impact for your program. – <https://www.cdc.gov/eval/logicmodels/index.htm>
 - See the diagram on page 6 of https://www.cdc.gov/dhds/docs/logic_model.pdf
 - <https://journals.plos.org/plosmedicine/article?id=10.1371/journal.pmed.0020124>
 - <https://www.biorxiv.org/content/10.1101/050575v1.full.pdf>
 - Meetings with client
- [1] Christian Collberg and Todd A. Proebsting. 2016. Repeatability in computer systems research. *Communications of the ACM* 59, 62–69. DOI:<https://doi.org/10.1145/2812803>
 - [2] Nasir U. Eisty and Jeffrey C. Carver. Developers Perception of Peer Code Review in Research Software Development.
 - [3] Simon Hettrick. A journey of reproducibility from Excel to Pandas. Retrieved from <https://www.software.ac.uk/blog/2017-09-06-journey-reproducibility-excel-pandas>
 - [4] Simon Hettrick, Mario Antonioletti, Les Carr, Neil Chue Hong, Stephen Crouch, David De Roure, Iain Emsley, Carole Goble, Alexander Hay, Devasena Inupakutika, Mike Jackson, Aleksandra Nenadic, Tim Parkinson, Mark I Parsons, Aleksandra Pawlik, Giacomo Peru, Arno Proeme, John Robinson, and Shoaib Sufi. 2014. UK Research Software Survey 2014. DOI:<https://doi.org/10.5281/zenodo.14809>
 - [5] Jayanti Bhandari Neupane, Ram P. Neupane, Yuheng Luo, Wesley Y. Yoshida, Rui Sun, and Phillip G. Williams. Characterization of Leptazolines A–D, Polar Oxazolines from the Cyanobacterium *Leptolyngbya* sp., Reveals a Glitch with the “Willoughby–Hoye” Scripts for Calculating NMR Chemical Shifts. *Organic Letters* 21. DOI:<https://doi.org/10.1021/acs.orglett.9b03216>
 - [6] A. F. Rosene, J. E. Connolly, and K. M. Bracy. Software Maintainability — What It Means and How to Achieve It. *IEEE Transactions on Reliability R-30*. DOI:<https://doi.org/10.1109/TR.1981.5221065>
 - [7] Patrick Vandewalle, Jelena Kovacevic, and Martin Vetterli. *IEEE Singal Processing Magazine* 26. DOI:<https://doi.org/10.1109/MSP.2009.932122>
 - [8] Patrick H. Willoughby, Matthew J. Jansma, and Thomas R. Hoye. A guide to small-molecule structure assignment through computation of (1H and 13C) NMR chemical shifts. *Nature Protocols* 9. DOI:<https://doi.org/https://doi.org/10.1038/nprot.2014.042>