Tools! Tools! We need tools!

Diomidis Spinellis

Athens University of Economics and Business

Tools in science

In 1908 Ernest Rutherford won the Nobel Prize in Chemistry “for his investigations into the disintegration of the elements, and the chemistry of radioactive substances”. In support of his candidacy the Nobel Committee for Chemistry wrote about the elegant experiments he performed to show that alpha particles were in fact doubly-charged helium atoms. Rutherford was able to show this through a simple but ingenious device. He had a glassblower create a tube with an extremely thin wall that allowed the alpha particles emanating from the radon gas it contained to escape. Surrounding that tube was another from which he had emptied the air. After some days he found that the material accumulated in the outer tube produced the spectrum of helium [1].

Science has always progressed mightily through the use of tools, which are increasingly designed by scientists but built by engineers and technicians. Telescopes allow us to see stars at the edge of our universe, imaging satellites uncover the workings of our Earth, genome sequencers and microscopes let us examine cells and molecules, and particle accelerators peer into the nature of atoms. Currently the world’s largest single machine is a tool explicitly built to advance our scientific understanding of matter: CERN’s 27km-round Large Hadron Collider, which more than ten thousand scientists and engineers from over a hundred countries built over a period of ten years.

The tools we need

The availability and use of large data sets associated with software development has transformed software engineering in ways described in other chapters of this book. A key element for the application of data science in software engineering is the availability of suitable tools. Such tools allow us to obtain data from novel sources, measure processes and products, and analyze all that data to derive insights that can advance science and everyday practice. By definition, scientific advancement happens through work beyond the state of the art, so it should come as no surprise that a lot of effort in data science involves building and refining tools. In the following paragraphs I outline important types of tools and best practices for building them. In order to provide insights on the building of tools, the description is mostly based on personal experience.

First we need tools for **obtaining metrics**. Although software metrics have been with us for decades, tools for obtaining them reliably are often hard to come by. I've seen research work where the metric collection was treated as an afterthought, apparently delegated to inexperienced undergraduate students. This is often evident from the quality of the corresponding tools, which may not scale, may produce erroneous results, or may be difficult to build upon.

Partly as a result of such problems in 2005 I built, ckjm a tool that derives Chidamber and Kemerer metrics from Java programs [2]. These are the weighted methods per class, the depth of the inheritance tree, the number of children per class, the coupling between object classes, the response for a class, and the lack of cohesion in methods [16]. Designing *ckjm* to work as a Unix-style filter allowed it to analyze arbitrarily-large projects, an advantage appreciated by many of its users.

Also during 2000-2010 I built CScout, a source code analyzer and refactoring browser for collections of C programs. It can process workspaces of multiple projects (a project is defined as a collection of C source files that are linked together) mapping the complexity introduced by the C preprocessor back into the original C source code files. *CScout* takes advantage of modern hardware advances (fast processors and large memory capacities) to analyze C source code beyond the level of detail and accuracy provided by current compilers, linkers, and other source code analyzers [3]. The analysis *CScout* performs takes into account the identifier scopes introduced by the C preprocessor and the C language proper scopes and namespaces. After the source code analysis *CScout* can * perform accurate cross project identifier renames, * process sophisticated queries on identifiers, files, and functions, * locate unused or wrongly-scoped identifiers, * identify header files that don't need to be included, and * create call graphs spanning both C functions and function-like macros. The implementation of *CScout* required developing a theory behind the analysis of C code in the presence of the preprocessor [4], and the detailed handling of many compiler extensions and edge cases. I used *CScout* to compare four operating system kernels [5] and later look at the optimization of header-file include directives [6]. Both tasks required months of work in order to adjust *CScout* to the requirements of the analysis task. Despite its sophistication *CScout* has seen considerably less use than *ckjm*, probably because of the considerable work required to put it to work.

More recently, in order to analyze the use and evolution of C language constructs and style I adopted a simpler approach, and built qmcalc: a tool that will perform lexical analysis of C source code read from its standard input and calculate and print numerous metrics associated with it. The program reads a single C file from its standard input and outputs raw figures and diverse quality metrics associated with the code. These include the number of functions, lines, and statements; the number of occurrences of various keywords; the use of comments and preprocessing; the number and length of identifiers; the Halstead and cyclomatic compexity per function; the use of spacing for indentation; a measure of style inconsistency; and numbers associated with probable style infractions. What *qmcalc* lacks in sophistication it offers in versatility, as it can process any code thrown at it, including code with errors or obscure undocumented constructs. This made it easy to analyze millions of lines of diverse code [7].

A second category of tools are those we use to **obtain or synthesize data** from processes and running products, which can then be distilled into metrics. Such tools bridge the gap between the utilitarian data formats used to support software developers and the needs of data science for software engineering. Given that computers are reflective machines the possibilities for data collection are endless. One example in this category is GHTorrent, a system that obtains data through GitHub's event API (whose raison d'être is the automation of software development processes) and makes them available as a database [8, 17]. Another is a set of tools used to synthesize a Git repository containing 44 years of Unix evolution from software distribution snapshots and diverse configuration management repositories [9]. The development of both tools demonstrated the difficulties associated with processing big, incomplete, and fickle data sets. The associated tools must be able to handle perverse cases, such as dates lying several years into the future or several kilobytes long file names. Other interesting data generation tools are those that instrument IDEs to obtain usage details [10]. These can give us valuable insights on how developers actually work, minimizing the risk of self-report bias. Instrumenting programs, libraries, and middleware can also provide valuable data. For example, by modifying memory allocation functions and a call graph profiler's function call processing code I obtained data to illustrate memory fragmentation and stack size variability [11].

Finally, a third category of tools are those we use to **analyze data**. Thankfully in this segment there are many general purpose mature tools and libraries that we can readily use. These include R, Python's data tools, and relational database management systems often (mis)used to perform online analytical processing. Skimping on the effort required to master these tools in favor of ad hoc approaches is a mistake. Then, there are also specialized platforms, such as *Alitheia Core* [12], _Evolizer_ [13], and *Tesseract* [14], that can analyze software engineering data. These can be very helpful if the research question matches closely the tool's capabilities. Otherwise, their complexity often makes tailoring them more expensive than developing bespoke tooling.

Recommendations for tool building

Given the importance of tools in conducting software engineering research, the most important piece of advice is to **hone your tool-building skills**. I have written tools in Perl, the Unix shell, C++, and Java. C++ can be beneficial when extreme performance is required (in some cases I have run processing jobs that took many months to complete). Java can be useful when interacting with other elements in its ecosystem, for example the Eclipse platform. Perl has the advantage of a huge library of mature components that can cover even the most specialized needs, such as processing legacy *source code control system* (SCCS) files, but the underlying language shows its age. Using the Unix shell benefits from the power of the hundreds of tools available under it, and can be a particularly good choice when the heavy lifting will be performed by such tools. Otherwise, a modern scripting language, such a Python or Ruby, can offer the best balance between versatility, programmer efficiency, and performance. Choose the language that appeals to your taste and requirements, and sharpen your skills in its use.

Given that many of the tools used are bespoke contraptions rather than mature software, **testing** them thoroughly is a must. Thankfully, the practice of unit testing provides methods for performing this task in an organized and systematic fashion. According to the software's change logs, when developing *qmcalc* 130 unit tests uncovered more than 15 faults. Without these tests some of these faults might have resulted in erroneous results when the tool was used. Given the large data sizes processed, testing can often be optimized through appropriate sampling. This allows the data input and output to be carefully inspected by hand in order to validate the tool's operation.

Finally, when developing tools consider **sharing** the results of your efforts as open source software and contributing to other similar endeavors. This allows our field to progress by standing on each other's shoulders rather than toes. It is also a practice that aids the reproducibility of research, as others can easily obtain and reuse the tools used for conducting it. In addition, the knowledge that your tool will be shared as open source software where the whole world will be able to see and judge it, puts pressure on you to develop it from the beginning, not as a quick and dirty throwaway hack, but as the high-quality piece of software it deserves to be.

Historians have commented that when Rutherford's glassblower, Otto Baumbach, was interned during the First World War, experimental physics at the University of Manchester where he had set up shop were brought to a halt [15]. Such is the power of tools to advance great science.

References

1. Rutherford, E. and Royds, T. The Nature of the alpha particle from Radioactive Substances, *Philosophical Magazine*, 17(6):281-286, 1909.
2. Diomidis Spinellis. Tool writing: A forgotten art? *IEEE Software*, 22(4):9–11, July/August 2005. (doi:10.1109/MS.2005.111).
3. Diomidis Spinellis. CScout: A refactoring browser for C. *Science of Computer Programming*, 75(4):216–231, April 2010. (doi:10.1016/j.scico.2009.09.003)
4. Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, November 2003
5. Diomidis Spinellis. A tale of four kernels. In Wilhem Schafer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 381–

390, New York, May 2008. Association for Computing Machinery. (doi:10.1145/1368088.1368140).
6. Diomidis Spinellis. Optimizing header file include directives. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(4):233–251, July/August 2009. (doi:10.1002/smr.369)
7. Diomidis Spinellis, Panagiotis Louridas, and Maria Kechagia. An exploratory study on the evolution of C programming in the Unix operating system. In *ESEM '15: 9th International Symposium on Empirical Software Engineering and Measurement*, pages 54–57. IEEE, October 2015.
8. Georgios Gousios and Diomidis Spinellis. GHTorrent: Github's data from a firehose. In Michele Lanza, Massimiliano Di Penta, and Tao Xie, editors, *9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE, June 2012. (doi:10.1109/MSR.2012.6224294)
9. Diomidis Spinellis. A repository with 44 years of Unix evolution. In MSR '15: Proceedings of the 12th Working Conference on Mining Software Repositories, pages 13–16. IEEE, 2015. (doi:10.1109/MSR.2015.6)
10. Murphy, Gail C., Mik Kersten, and Leah Findlater. How are Java software developers using the Elipse IDE? *IEEE Software*, 23.4 (2006): 76–83.
11. Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison-Wesley, Boston, MA, 2006.
12. Georgios Gousios and Diomidis Spinellis. Conducting quantitative software engineering studies with Alitheia Core. *Empirical Software Engineering*, 19(4):885–925, August 2014. (doi:10.1007/s10664-013-9242-3)
13. Gall, Harald C., Beat Fluri, and Martin Pinzger. Change analysis with evolizer and changedistiller. *IEEE Software* 26(1):26–33, January-February 2009.
14. Sarma, Anita, et al. Tesseract: Interactive visual exploration of socio-technical relationships in software development. *ICSE 2009: IEEE 31st International Conference on Software Engineering.* IEEE, 2009.
15. Gall, Alan. Otto Baumbach — Rutherford's Glassblower. *Newsletter published by History of Physics Group of the Institute of Physics (UK & Ireland)*, (23):44–55, January 2008.
16. Chidamber, S. R. and Kemerer, C. F., A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering*, vol. 20, 1994.
17. Gousios, G. The GHTorrent dataset and tool suite. In MSR '13: Proceedings of the 10th Working Conference on Mining Software Repositories, pages 233–236. 2013.

Note