Knowledge is Power (and Everything Else) Supporting Knowledge-Centric Software Engineering

Dan Szymczak[1 [0000−0002−9773−0602]],[0000−0000−0000−0000],[0000−0000−0000−0000], and[0000−0000−0000−0000]

McMaster University, Hamilton, ON, Canada,
szymczdm@mcmaster.ca

Abstract. The abstract should summarize the contents of the paper using at least 70 and at most 150 words. It will be set in 9-point font size and be inset 1.0 cm from the right and left margins. There will be two blank lines before and after the Abstract. ...

Keywords: computational geometry, graph theory, Hamilton cycles

1 Introduction

Software today is developed on varying scales with differing scopes and aims. We see small-scale software, including one-off scripts, designed for a single purpose with a small user-base in mind. Business and large-scale software solutions are being used around the world with a massive user-bases. Safety-critical, certified software is also found globally with a user base in far more specialized fields than business software, yet the user-base are not the only ones affected by the software. Consider nuclear power control or medical imaging systems, they have the potential to affect many more people than those that actually use the software, hence why they must be certified and maintained at the highest quality.

In the Scientific Computing (SC) community, we see the use of software at many scales, with many different quality standards. Regardless of scale, software can, and should, be developed to the highest possible standard. However, the overall quality of many projects, tends to suffer due to external factors including, but not limited to: the size of a development team, budget constraints, and project deadlines.

It is our view that software quality shouldn't be so wholly dependent on the size of the development team or scale of the project. We (as Software Engineers) should be working to bridge this quality gap by providing tools to facilitate easier creation of maintainable, reusable, and certifiable software. With that in mind, how can we accurately measure the quality of our software?

1.1 Quality Metrics

- Finding quality metrics is difficult - similar to finding metrics for productivity. Many have been proposed, but few correlations found after data analysis.- Let's

stay away from “fuzzy” concepts that we can’t quite quantify. - Use traceability as a common-sense metric for maintainability. - Reuse can be measured through number of projects, though it isn’t perfect (ie. highly specialized code may be reusable, but is only used in a small number of highly specialized projects) - Certifiability is simple to quantify (theoretically), but costly to test in practice (ie. put your software through the certification process. The number of software certified compared to those developed = our metric.

1.2 The case for documentation

- Documentation goes hand in hand with quality – we can’t measure things like traceability without it.

- Bring up re-dev case studies and problems with certification therein / other issues with developing the software with all documentation (time, etc.) - Standard dev models (waterfall) are cumbersome, but faking them is useful []

- Many software projects (in SC) share the same base knowledge, but are often reimplemented by lone developers due to a lack of sufficient documentation or availability of others’ software. - Knowledge duplication (from textbooks or across software projects) is bad! Stop it! Even if its necessary sometimes due to poor documentation.

1.3 Proposed Solution

- What should we do? - Let’s take a different, more robust approach, with an aim at improving qualities along the way. - We should allow devs to focus on the underlying scientific knowledge as opposed to the SE/CS stuff. - Use advances made in generation techniques and tools to take advantage of captured/duplicate knowledge, not only from SE, but also from Scientific fields. - ̂forms basis of our knowledge-centric approach.

2 Knowledge-Centric Software Engineering

- Focus on underlying knowledge - Capture knowledge in a meaningful way - Use it wherever necessary while maintaining full traceability (one-source) - Transform knowledge as necessary for intended artifacts - FIG: Single source -¿ multiple artifacts; Easy to trace - Expedites prog family generation

2.1 Capturing Knowledge

- Many different types of knowledge (theories, expressions, concepts, etc.) - Need to capture all information around a piece of knowledge, even if it may not seem relevant at this point in time. - Captured knowledge is kept in a knowledge-base to be used inter- and intra-project - Essentially building ontologies for specific disciplines over time

2.2 Operationalizing Knowledge

- Knowledge base can be reused (touched on above) - Use of Recipes – create artifacts (views) of the knowledge - Transforms knowledge into a relevant form (equations vs. executable code vs. English language descriptions) - Program families can be built quickly – change decisions for one member to create another.

3 Tool Support with Drasil

- For KCSE to be viable, we need a strong support framework. - Brief intro to Drasil - What is it / what can it do? Generate all the things! (Docs / Code) - KC in Drasil - Recipes - Key Components (give brief intro to expand on later – in 3.2)

3.1 A Grounded Theory – Drasil's development

- Development following a Grounded Theory (cite terminology?) ie. Practical approach - Go into detail a la SE-CSE paper

3.2 Drasil Today

- Talk about the design and the neat things from notes.txt - FIG: Chunk class hierarchy - Little languages - Generator - Gool incorporation? - Sentences - Handling some of English's weirdness for generation (NP) - etc.

4 GlassBR – A Case Study

- Introduce case studies and reimplementation stuff - Focus on GlassBR to start (as it is most complete); we will also discuss interesting findings from the other case studies afterwards

4.1 Knowledge

- FIG: SI_Units - FIG: Specific chunks example

4.2 SRS

- FIG: Little language spec of SRS - Walk through some of the generation aspect of the little lang?

4.3 Code

- GOOL stuff (might need a hand from Steven here)

5 Interesting Results from Other Case Studies

- Each case study had some issue arise during reimplementation with Drasil.

5.1 Common Across Projects

- Implementing symbol table revealed missing symbols - Tedious (for human) sanity checks are 'free'; happen on every run of generator - Found many errors involving symbols. - Units can be checked to ensure they are consistent - Constraints can be automatically guarded against

5.2 SWHS

- Program family (SWHS and NoPCM) - Much of NoPCM is imported (directly!) from SWHS - Found some interesting errors - Implementing symbol table revealed missing symbols; - extraneous PCM symbols in NoPCM due to IM sharing

5.3 SSP

- Symbols for quantities changed throughout the documentation - Never noticed by a human! Found almost instantly by Drasil - new Symbols were undefined.

6 Future Work

7 Conclusion

References

1. Ackroyd, K.S., Kinder, S.H., Mant, G.R., Miller, M.C., Ramsdale, C.A., Stephenson, P.C.: Scientific software development at a research facility. IEEE Software 25(4), 44–51 (July/August 2008)
2. Beyak, L., Carette, J.: SAGA: A DSL for story management. In: Danvy and chieh Shan [7], pp. 48–67
3. Bowman, M., Debray, S.K., Peterson, L.L.: Reasoning about naming systems. ACM Trans. Program. Lang. Syst. 15(5), 795–825 (November 1993)
4. Braams, J.: Babel, a multilingual style-option system for use with latex's standard document styles. TUGboat 12(2), 291–301 (June 1991)
5. Carver, J.C., Kendall, R.P., Squires, S.E., Post, D.E.: Software development environments for scientific and engineering software: A series of case studies. In: ICSE '07: Proceedings of the 29th international conference on Software Engineering. pp. 550–559. IEEE Computer Society, Washington, DC, USA (2007)
6. Clark, M.: Post congress tristesse. In: TeX90 Conference Proceedings. pp. 84–89. TeX Users Group (March 1991)
7. Danvy, O., chieh Shan, C. (eds.): Proceedings IFIP Working Conference on Domain-Specific Languages, EPTCS, vol. 66 (2011)
8. Dubois, P.F.: Designing scientific components. Computing in Science and Engineering 4(5), 84–90 (September 2002)
9. Easterbrook, S.M., Johns, T.C.: Engineering the software for understanding climate change. Comuting in Science & Engineering 11(6), 65–74 (November/December 2009)
10. Hatcliff, J., Heimdahl, M., Lawford, M., Maibaum, T., Wassyng, A., Wurden, F.: A software certification consortium and its top 9 hurdles. Electronic Notes in Theoretical Computer Science 238(4), 11–17 (2009), http://linkinghub.elsevier.com/retrieve/pii/S1571066109003570
11. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Trans. Program. Lang. Syst. 15(5), 745–770 (November 1993)
12. Johnson, A., Johnson, B.: Literate programming using `noweb`. Linux Journal 42, 64–69 (October 1997)
13. Kelly, D.: Industrial scientific software: A set of interviews on software development. In: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research. pp. 299–310. CASCON '13, IBM Corp., Riverton, NJ, USA (2013), http://dl.acm.org/citation.cfm?id=2555523.2555555
14. Kelly, D.: Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. Journal of Systems and Software 109, 50–61 (2015), http://dx.doi.org/10.1016/j.jss.2015.07.027
15. Knuth, D.E.: Literate programming. The Computer Journal 27(2), 97–111 (1984), http://comjnl.oxfordjournals.org/content/27/2/97.abstract
16. Lamport, L.: LaTeX User's Guide and Document Reference Manual. Addison-Wesley Publishing Company, Reading, Massachusetts (1986)
17. Nedialkov, N.S.: VNODE-LP — a validated solver for initial value problems in ordinary differential equations. Tech. Rep. CAS-06-06-NN, Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 4K1 (2006)
18. Pharr, M., Humphreys, G.: Physically Based Rendering: From Theory to Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)

19. Roache, P.J.: Verification and Validation in Computational Science and Engineering. Hermosa Publishers, Albuquerque, New Mexico (1998)
20. Salas, S., Hille, E.: Calculus: One and Several Variable. John Wiley and Sons, New York (1978)
21. Segal, J.: When software engineers met research scientists: A case study. Empirical Software Engineering 10(4), 517–536 (October 2005), http://dx.doi.org/10.1007/s10664-005-3865-y
22. Segal, J.: Models of scientific software development. In: Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering (SECSE 2008). pp. 1–6. In conjunction with the 30th International Conference on Software Engineering (ICSE), Leipzig, Germany (2008), http://www.cse.msstate.edu/ SECSE08/schedule.htm
23. Segal, J., Morris, C.: Developing scientific software. IEEE Software 25(4), 18–20 (July/August 2008)
24. Smith, S., Koothoor, N.: A document driven method for certifying scientific computing software used in nuclear safety analysis. Nuclear Engineering and Technology Accepted (2016), 42 pp
25. Szymczak, D.: Generating Learning Algorithms: Hidden Markov Models as a Case Study. Master's thesis, McMaster University, Hamilton, ON, Canada (2014)
26. Wilson, G., Aruliah, D., Brown, C.T., Hong, N.P.C., Davis, M., Guy, R.T., Haddock, S.H., Huff, K.D., Mitchell, I.M., Plumblet, M.D., Waugh, B., White, E.P., Wilson, P.: Best practices for scientific computing. CoRR abs/1210.0530 (2013)
27. Yu, W.: FASCS: A Family Approach for Developing Scientific Computing Software. Ph.D. thesis, McMaster University, Hamilton, ON, Canada (2011)