banner above paper title

Title Text

Subtitle Text, if any

Dan Szymczak
McMaster University
szymczdm@mcmaster.ca

Jacques Carette Spencer Smith
McMaster University
carette@mcmaster.ca / smiths@mcmaster.ca

Abstract

This will be filled in once we're done drafting. For now I'm putting in a solid block of text that will let me ramble on for a while when it comes down to the final abstract. Programmers are like gods of tiny universes. They create worlds according to their rules. If we think of software artifacts (that is, all documentation, build instructions, source code, etc.) as their own tiny worlds, a universe takes shape. Now think about what would happen if the laws of that universe were inconsistent between worlds. What if the laws of thermodynamics fundamentally changed as soon as you left Earth? Say goodbye to space travel. Keeping artifacts consistent with each other is hugely important in the software world, and Drasil aims to ensure it...

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords Literate software, knowledge capture, traceability, software engineering,scientific computing, artifact generation, software quality

1. Introduction

We have a vision to make better software. We want to increase software's maintainability, traceability, reproducibility, verifiability, and reusability. Our vision is to invest more in the short-term to provide outstanding long-term benefits and is where we see the fundamental trade-off in our technique: it is the complete opposite of agile-development while simultaneously throwing away the waterfall model.

We intend to make working on program families trivial for the cost of losing the capability to create "local hacks" to "just get things working".

Before we get into more detail on the technique, let's start with an obvious disclaimer: it would be a pipe dream to claim we could use this technique over all software domains. In fact, the domain(s) we will be looking at are very restrictive, but highly useful. We have restricted the scope of our work to only those areas of software with a *well understood* knowledge base.

We will be sticking to using Scientific Computing (SC) software for illustrative purposes throughout this paper, but the fundamental ideas (and, in fact, our framework) should be applicable to any well understood software domain.

We believe Literate Programming (LP) had the right idea, but in our opinion is far too restricted. Rather than just coupling the source code with its documentation, we want to build all of our software artifacts (requirements and design documents, source files, test cases, build instructions, etc.) from a singular common knowledge base.

We want to break away from the LP idea of "one source", instead introducing the idea of "common knowledge" libraries for problems that come up regularly. For example: the theory behind heat transfer including the conservation of energy equations would be in a "common knowledge" library. We could also see the idea of the Système International (SI) Units as another piece of "common knowledge".

We also very much want to keep the idea of "chunks" and assemble everything from chunks. How we do that will be discussed in more detail in Section **??**.

Another disclaimer before we go further: none of the ideas presented in this paper are new. We are building on a very longstanding history of work, and not being particularly creative with the ideas therein. Knuth himself even said that "I have simply combined a bunch of ideas that have been in the air for a long time" when he coined LP (**?**). We are, however, taking old ideas and breaking them down into something practical that we can work with. We want to *do* something, not spend years stuck in the design phase.

Others have done similar work (which we will get into in Section **??**), but they did not achieve the results we are envisioning. They either set their aims on other targets, spent too long creating a grand design and ended up without any real, practical results, or they simply were not brave enough to break things down into the smallest necessary chunks.

We believe that we have assembled the right ideas to achieve our vision. The overall success or failure of our approach hinges on the idea of a stable, well understood knowledge base. Without that we might as well give up and go home. If we do not understand the fundamentals behind the software we intend to create, then we will be unable to properly encapsulate that fundamental knowledge for use in our framework (effectively getting us nowhere).

As it stands, we are on a (not so) humble, practical route to achieving our goals and improving the overall quality of software.

2. Drasil

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984. doi: 10.1093/comjnl/27.2.97. URL `http://comjnl.oxfordjournals.org/content/27/2/97.abstract`.