A Sustainable Approach to Developing Software in Well Understood Domains

Jacques Carette
Computing and Software
McMaster University
Hamilton, Ontario, Canada
carette@mcmaster.ca

Spencer Smith
Computing and Software
McMaster University
Hamilton, Ontario, Canada
smiths@mcmaster.ca

ABSTRACT

Missing abstract.

CCS CONCEPTS

• **Theory of computation → Abstract machines**; • **Software and its engineering** → *Application specific development environments*; *Requirements analysis*; *Specification languages*; **Automatic programming**.

KEYWORDS

code generation, document generation, knowledge capture, software engineering, scientific software

ACM Reference Format:
Jacques Carette and Spencer Smith. 2022. A Sustainable Approach to Developing Software in Well Understood Domains. In *Pittsburgh '22: NIER - New Ideas and Emerging Results (ICSE 2022), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

"Software" is not uniform. To use the exact same process for developing an embedded safety-critical piece of code (like that of a pacemaker), the flight control software for an airplane, a one-off script for moving some files around, and some amusing animations on one's personal web site, is patently ridiculous.

The same is true in say, civil engineering: you don't need architects, licensed engineers and a million permits to build a small shed in your backyard, but you do need them to build a 100 story skyscraper.

Which brings us to our central topic: there are some kinds of software where our current development methods *are all wrong*. Our task is to define exactly which type of software we have in mind, and then derive an entirely different development methodology that is customized to the special characteristics of that strict subset.

There are many properties of software that can be used for providing a classification. Here we will focus on one particular "axis": how **well understood** it is. The majority of the next section will be devoted to explaining exactly what this means. Once that is set up, we can then unravel some operational consequences: how the characteristics of well understood softare lead to innovative methods of building such software. As this might be perceived as too abstract, we give a very concrete example. Of course, our ideas do not exist in a vacuum: we were inspired by a number of connected ideas, and we then give credit where credit is due. More than just ideas, there are also technologies that back these ideas, some of which we're already using, others which lie in our future, and we outline some of these as well.

2 WHAT IS "WELL UNDERSTOOD" SOFTWARE?

Definition 1. *A software domain is* well understood *if*

(1) the domain knowledge (DK) is codified,
(2) the computational interpretation of the DK is clear,
(3) the engineering of code to perform said computations is well understood.

By *codified*, we mean that the knowledge exists in standard form in a variety of textbooks. For example, many domains of knowledge in engineering use differential equations as models. Furthermore, the quantities of interest are known, given standard names and standard units. In other words, there is an established vocabulary and body of knowledge that is uncontroversial.

We can further refine these high level ideas as follows, where we use the same numbering as above to indicate which part of the definition is being directly refined, but where the refinement nevertheless should be understood more holistically.

(1) Models in the DK *can be* written formally.
(2) Models in the DK *can be* turned into functional relations by existing mathematical steps.
(3) Turning these functional relations into code is also an understood transformation.

Perhaps the most important aspect of this refinement is that the last two parts deeply involve *choices*: What quantities are considered inputs, outputs and parameters to make the model functional? There are also a host of choices, including which programming language, but also software architecture, data-structures, algorithms, etc, which are also part of creating the code.

It is important to understand that *well understood* does not imply *choice free*. Writing a small script to move some files around can be easily written as a Shell script, or in Python or in Haskell, depending on the author's style. In all cases, assuming the author chooses a language in which they are fluent, the job will be entirely straightforward.

Lest our reader gets misled into thinking that code is the only artifact that matters, we should explicitly refine our definition in a different direction, equally important.

(1) The meaning of the models is understood at a human-pedagogical level, i.e. it is explainable.
(2) Combining models is also explainable. Thus the *transformers* we mentioned before simultaneously operate on mathematical representations and on explanations. This requires that English descriptions also be captured in the same manner as the formal-mathematical knowledge.
(3) Similarly, the *transformers* the arise from making software oriented decisions requires that they be captured with a similar mechanism, including English explanations as well.

We dub these *triform theories*, as a nod to *biform theories*[**?**]. The idea is that we couple (1) an axiomatic description, (2) a computational description, and (3) an English description of a concept.

It is important to notice that there are various kinds of choices embedded in the different kinds of knowledge. They can show up simply as *parameters*, for example the gravity constant associated to a planet. This also shows up as different transformers, for example turning $F - m \cdot a = 0$ into $F(m, a) = m \cdot a$, i.e. from a conservation law into a computation. Note that, for motion computation, that same conservation law is often rewritten as $a(m, F) = F/m$ as part of solving $a = \ddot{x}$ to obtain a position (x) as a function of time (t). And we also get choices of phrasing, which are equivalent but may be more adequate in context, for example.

3 HOW WOULD YOU GO BUILDING THAT?

So what would be a reasonable process for building a piece of software assuming some kind of infrastructure exists for recording the kind of knowledge outlined in section 2? Let us outline a chronological "story" of such an idealized process. It is important to note that we're **not** outlining the process to follow, but rather the *idealized process* (influenced by Parnas' [2]).

(1) Have a problem to solve, or task to achieve, which falls into the realm where *software* is likely to be the central part of the solution.
(2) Convince oneself that the underlying problem domain is *well understood*, as defined in section 2.
(3) Describe the problem:
    (a) Find the base knowledge (theory) in the pre-existing library or, failing that, write it if it does not yet exist,
    (b) Assemble the ingredients into a coherent narrative,
    (c) Describe the characteristics of a good solution,
    (d) Come up with basic examples (to test correctness, intuitions, etc).
    (e) Identify the naturally occurring known quantities associated to the problem domain, as well as some desired quantities. For example, some problems naturally involve lengths lying in particular ranges, while others will involve ingredient concentrations, again in particular ranges.
(4) Describe, by successive transformations, how the natural knowledge can be turned from a set of relations (and constraints) into a deterministic[1] input-output process.
    (a) This set of relations might require *specializing* the theory (eg. from n-dimensional to 2-dimensional, assuming no friction, etc). These *choices* need to be documented, and are a crucial aspect of the solution process. The *rationale* for the choices should also be documented. Lastly, whether these choices are likely or unlikely to change in the future should be recorded.
    (b) This set of choices is likely dependent, and thus somewhat ordered. In other words, some *decisions* will enable other choices to be made that would otherwise be unavailable. Eg: some data involved in the solution process is orderable, so that sorting is now a possibility that may be useful.
(5) Describe how the computation process from step 4 can be turned into code. Note that the same kinds of choice can occur here.
(6) Turn the steps (i.e. from items 4 and 5) into a *recipe*, aka program, that weaves together all the information into a variety of artifacts (softifacts). These can be read, or execute, or … as appropriate.

While this last step might appear somewhat magical, it isn't. The whole point of defining *well understood* is to enable that last step. A suitable knowledge encoding is needed to enable it, but this step is a reflection of what humans currently themselves do when assembling these very same softifacts. We are merely being explicit about how to go about mechanizing these steps.

What is missing is an explicit *information architecture* of each of the necessary softifacts. In other words, what information is necessary to enable the mechanized generation of each softifact? It turns out that many of them are quite straightforward.

It is worthwhile to note that way too many research projects skip step 1 and 3: in other words, they never really write down what problem they're trying to solve. This is part of the **tacit knowledge** of a lot of research software. It is crucial to our whole process that this knowledge go from tacit to explicit. This is also one of the fundamental recognitions of *Knowledge Management* [**?**].

TODO: insert graphical illustration of the funnel from information to softifacts.

4 EXAMPLE

5 CONNECTED IDEAS

A multitude of ideas, old and new, have influenced us. They appear indirectly in our work; we use the conceptual aspects, and not the associated technologies (when they exist). The technologies we do use are in Section 8.

We do not always use the original conceptualization of the work, but a modern re-interpretation, incorporation various "lessons learned" through years of use. To keep things short, we outline our take-away, and refer the reader to the original literature for the initial view of the ideas.

[1]For the moment, we explicitly restrict our domain to deterministic solutions, as a meta-design choice. This can be expanded later.

6 RE-ORGANIZING ARTIFACTS

The most important idea that got all of this started is *literate programming*[1]. A key observation here is that computer code really has two audiences: the computer, proxied via a compiler or interpreter, and human readers. Traditionally, code is arranged for the convenience of the compiler[2]. Many languages are quite inflexible with respect to how code must be arranged to be acceptable. This directly clashes with the desire to inform the human reader of the code as to the *underlying story* that forms the backbone of why the code is written the way it is.

Thus the fundamental ideas of literate programming are:

(1) The idea of not writing the eventual artifacts directly,
(2) Of being able to "chunk up" pieces of code in arbitrary ways,
(3) Of writing a program to explicitly weave together the chunks into a final program.

In the end, two artifacts are generated: a human-readable "story" that is nicely typeset, that follows a logical flow amenable to human understanding, and a piece of code.

Not all literate programming tools actually support all these features. For example *literate Haskell* only keeps the feature of nice typesetting, completely dropping the 3 main ideas above!

The next source of idea is *org-mode*[**?**]. A shallow view is that org-mode is an Emacs major mode for plain text markup[3] *and much more*. It is that "and much more" which is of interest here. Amongst other features, Org-mode lets you write documents that mixes many different languages together. It also allows you to run certain code blocks and insert their results into the document itself. And, like literate programming, it also allows the export of both nice documents (via LaTeX) and code (via a tangling process). The paper *A Computing Environment for Literate Programming and Reproducible Research*[3] further describes the features and process. The possibilities are quite extensive.

A slightly different take on similar ideas is offered by *Jupyter notebooks* and *JupyterLab*[4]. To quote the developers' own descriptions:

> The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

and

> JupyterLab is a web-based interactive development environment for Jupyter notebooks, code, and data. JupyterLab is flexible: configure and arrange the user interface to support a wide range of workflows in data science, scientific computing, and machine learning. JupyterLab is extensible and modular: write plugins that add new components and integrate with existing ones.

[2]we use "compiler" even though this applies equally well to interpreters
[3]taken literally from https://orgmode.org.
[4]https://jupyter.org

The principal difference is that the main interface is a web page which is furthermore *very interactive*. It is also multi-lingual. The weaving and tangling features, while somewhat present, are de-emphasized over interactivity. The all-in-one interactive document is the most important element. The feature set is very appealing, and contributes greatly to its success.

Commentary: The main weakness in all of these approaches is that they are all centered on a single-document idea. The information contained in the document itself is not re-usable. So while all three ideas are a definite improvement over more traditional means of doing development, it is still not enough. Furthermore, all three approaches still involve hand-writing a lot of code, even though that code is somewhat liberated from the strictures imposed by the languages themselves.

7 THE REST

- cognitive work analysis, ecological interface design
- knowledge management
- ontologies, domain knowledge
- biform theories
- variabilities and commonalities, program families, software product lines.
- re-use
- views (software architecture)
- software artifacts
- (re)certification
- some ities: traceability, consistency
- reproducible research
- knowledge-based SE?
- MDD, MDE
- Grounded Theory

8 SOME USEFUL TECHNOLOGIES

- DSLs
- code generation
- program families
- grammatical framework
- plate, multiplate, optics
- Problem Solving Environments (PSEs).
  Another item to consider adding to the list is Problem Solving Environments (PSE). A PSE is "[a] system that provides all the computational facilities necessary to solve a target class of problems. It uses the language of the target class and users need not have specialized knowledge of the underlying hardware or software" (Kawata et al., 2012). (From https://www.igi-global.com/dictionary/computer-assisted-problem-solving-environment-pse/41954). This is a different way to solve the same problem we are trying to solve. They want the user to work with their domain knowledge, but they accomplish this (as far as I can tell) by providing powerful general purpose tools and a language to interface with these tools. They are focusing on run-time variability, while we can focus on build time variability. An argument could be made that general purpose tools are overwhelming for people. Being able to generate an application that is as complex as the user needs, but no more

complex, sounds like a good thing to me. I believe they try to handle the complexity by often providing a graphical DSL.

The idea of PSEs might be getting old. My quick search didn’t find many newer papers. This review article from 2010 might be useful:
https://www.researchgate.net/publication/220147479_Review_of_PSE_Problem_Solving_Environment_Study

REFERENCES

[1] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97 arXiv:http://comjnl.oxfordjournals.org/content/27/2/97.full.pdf+html

[2] David Lorge Parnas and Paul C Clements. 1986. A rational design process: How and why to fake it. *IEEE transactions on software engineering* 2 (1986), 251–257.

[3] Eric Schulte, Dan Davison, Thomas Dye, Carsten Dominik, et al. 2012. A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software* 46, 3 (2012), 1–24.