When Capturing Knowledge Improves Productivity

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

Jason Balaci
Computing and Software
McMaster University
Hamilton, Ontario, Canada
balacij@mcmaster.ca

ABSTRACT

Current software development is often quite code-centric and aimed at short-term deliverables, due to various contextual forces. We're interested in contexts where different forces are at play. **Well understood domains** and **long-lived software** provide such an opportunity. By further applying generative techniques, aggressive knowledge capture has the real potential to greatly increase long-term productivity.

Key is to recognize that currently hand-written software artifacts contain considerable knowledge duplication. With proper tooling and appropriate codification of domain knowledge increasing productivity is feasible. We present an example of what this looks like, and the benefits (reuse, traceability, change management) thus gained.

CCS CONCEPTS

• **Software and its engineering** → *Application specific development environments*; *Requirements analysis*; *Specification languages*; **Automatic programming**.

KEYWORDS

code generation, document generation, knowledge capture, software engineering

ACM Reference Format:
Jacques Carette, Spencer Smith, and Jason Balaci. 2022. When Capturing Knowledge Improves Productivity. In *Pittsburgh '22: NIER - New Ideas and Emerging Results (ICSE 2022), May 21–29, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 THE CONTEXT

1.1 "Well understood" Software?

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

(1) its Domain Knowledge (DK) is codified,
(2) the computational interpretation of the DK is clear, and
(3) writing 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 engineering domains use ordinary differential equations as models, the quantities of interest are known, given standard names and standard units. In other words, standard vocabulary has been established over time and the body of knowledge is uncontroversial.

We can refine these high level ideas, using the same numbering, although the refinement 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 an understood transformation.

Most importantly, the last two parts deeply involve *choices*: What quantities are considered inputs, outputs and parameters to make the model functional? What programming language? What software architecture data-structures, algorithms, etc.?

In other words, *well understood* does not imply *choice free*. Writing a small script to move files could just as easily be done in Bash, Python or Haskell. In all cases, assuming fluency, the author's job is straightforward because the domain is well understood.

1.2 Long-lived software?

For us, long-lived software is software that is expected to be in continuous use and evolution for 20 or more years. The main characteristic of such software is the *expected turnover* of key staff. This means that all tacit knowledge about the software will be lost over time if it is not captured.

1.3 Productivity?

We adapt the standard definition of productivity [4], where inputs are labour, but adjust the outputs to be knowledge and user satisfaction, where user satisfaction acts as a proxy for effective quality . This explicit emphasis on all knowledge produced, rather than just the operationalizable knowledge (aka code) implies that human-reusable knowledge, i.e. documentation, should also be greatly valued.

1.4 Documentation

Our definition of well understood also applies to **documentation** aimed at humans. Explicitly:

(1) The meaning of the models is understood at a human-pedagogical level, i.e. it is explainable.
(2) Combining models is explainable. Thus *transformers* 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.

Pittsburgh '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

(3) Similarly, the *transformers* that arise from making software oriented decisions should be captured with a similar mechanism, and also include English explanations.

We dub these *triform theories*, as a nod to *biform theories* [6]. We couple (1) an axiomatic description, (2) a computational description, and (3) an English description of a concept.

1.5 Softifacts

Software currently consists of a whole host of artifacts: requirements, specifications, user manual, unit tests, system tests, usability tests, build scripts, READMEs, license documents, process documents, as well as code. We use the word *softifacts* for this collection.

Whenever appropriate, we use standards and templates for each of the generated artifacts. For requirements, we use a variant [10] of the IEEE [7] and Volere templates [9].

1.6 Examples of context

When are these conditions fulfilled? One example is *research software* in science and engineering. While the results of running various simulations is entirely new, the underlying models and how to simulate them are indeed well-known. One particularly long-lived example is embedded software for space probes (like Pioneer 10).

2 A NEW DEVELOPMENT PROCESS

Given appropriate infrastructure, what would be an *idealized process* (akin to Parnas' ideas of faking a rational design process [8])?

(1) Have a task to achieve where *software* can play a central part in the solution.
(2) The underlying problem domain is *well understood*.
(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, for instance the naturally occurring known quantities and associated constraints.
    (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).
(4) Describe, by successive refinement transformations, how the above can be turned into a deterministic[1] input-output process.
    (a) Some refinements will involve *specialization* (eg. from n-dimensional to 2-dimensional, assuming no friction, etc). These *choices* and their *rationale* need to be documented, as a a crucial part of the solution. Whether these choices are (un)likely to change in the future should be recorded.
    (b) Choices tend to be dependent, and thus (partially) ordered. *Decisions* frequently enable or reveal downstream choices.
(5) Describe how the process from step 4 can be turned into code. 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 (documentation, code, build scripts, test cases, etc). These can be read, or executed, or … as appropriate.

[1] A current meta-design choice.

While this last step might appear somewhat magical, it isn't. The whole point of defining *well understood* is to enable it! A *suitable* knowledge encoding is key. This is usually tacit knowledge that entirely resides in developers' heads.

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

Often steps 1 and 3 are skipped; this is part of the **tacit knowledge** of a lot of software. Our process requires that this knowledge be made explicit, a fundamental step in *Knowledge Management* [5].

3 AN EXAMPLE

We have built the needed infrastructure. It consists of 60KLoc of Haskell implementing a series of interacting Domain Specific Languages (DSLs) for knowledge encodings, mathematical expressions, theories, English fragments, code generation and document generation.[2] A full description would take too much space. Instead, we provide an illustrative example.

We will focus on information capture and the artifacts we can generate. For concreteness, we'll use a single example from our suite: GlassBR, used to assess the risk for glass facades subject to blast loading. The requirements are based on an American Standard Test Method (ASTM) standard [1, 2]. GlassBR was originally a Visual Basic code/spreadsheet created by colleagues in a civil engineering research lab. We added their domain knowledge to our framework, along with recipes to generate relevant artifacts. Not only can we generate code for the necessary calculations (in C++, C#, Java, Python and Swift), we added documentation that was not in the original (Software Requirements Specification, doxygen, README.md and a Makefile). Moreover, our implementation is actually a family of implementations, since some design decisions are explicitly exposed as changeable variabilities, as described below.

The transformation of captured knowledge is illustrated in Figure 1. This is read starting from the upper right box. Each piece of information in this figure has its own shape and colour (orange-cloud, pink lozenge, etc). It should be immediately clear that all pieces of information reappear in multiple places in the generated artifacts. For example, the name of the software (GlassBR) ends up appearing more than 80 times in the generated softifacts (in the folder structure, requirements, README, Makefile and source code). Changing this name would traditionally be extremely difficult; we can achieve this by modifying a single place, and regenerating.

The first box shows the directory structure of the currently generated softifacts; continuing clockwise, we see examples of Makefiles for the Java and Python versions, parts of the fully documented, generated code for the main computation in those languages, user instructions for running the code, and the processed LaTeXfor the requirements.

The name GlassBR is probably the simplest example of what we mean by *knowledge*: here, the concept "program name" is internally

[2] We will provide a link if the paper is accepted.

defined, and its *value* is used throughout. A more complex example is the assumption that the "Load Distribution Factor" (LDF) is constant (pink lozenge). If this needs to be modified to instead be an input, the generated software will now have LDF as an input variable. We also capture design decisions, such as whether to log all calculations, whether to in-line constants rather than show them symbolically, etc. The knowledge for GlassBR can also be reused in different projects.

We now give example encodings[3] corresponding to steps of the process.

Step 3a: Base Knowledge

A common idea prevalent in GlassBR is the different types of `Glass` (a canonical representation of glass):

UID	Term (Name)	Abbrev.	Domain
`fullyT`	Fully Tempered	FT	`[Glass]`
`heatS`	Heat Strengthened	HS	`[Glass]`
`iGlass`	Insulating Glass	IG	`[Glass]`
`lGlass`	Laminated Glass	LG	`[Glass]`
`glassTypeFac`	Glass Type Factor	GTF	`[Glass]`

We also need to capture the *data definition* of "Risk of Failure".

Label	Risk of Failure
Equation	$B = \frac{k}{(ab)^{m-1}}(Eh^2)^m LDFe^J$
Description	B is the Risk of Failure (Unitless) k is the surface flaw parameter $(\frac{m^{12}}{N^7})$ a & b are the plate length & width (m) ...
Source	[1], [2]

Step 3b: Coherent Narrative

The natural descriptions in GlassBR are produced using an experimental language for describing relations between knowledge. For example, the goal of GlassBR ("Predict-Glass-Withstands-Explosion") is to "Analyze and predict whether the *glass slab* under consideration will be able to withstand the **explosion** of a certain **degree** which is calculated based on *user input*.", where italicized names are "named chunks" (named ideas), and bold-faced names are "concept chunks" (named ideas with a domain of related ideas). We call this goal a "concept instance" (a concept chunk applied in some way). Through this experimental language, we are able to perform all sorts of static analysis on our artifacts.

Step 3c: Characteristics of a Good Solution

One of our outputs is a probability, which can be checked to be between 0 and 1 (not shown).

Step 4a: Specialization of Theories

To illustrate the specialization of theories, we will look outside of the GlassBr example because, due to its phenomenological nature, GlassBr only has trivial specializations. To illustrate this step we will turn to another example: a solar water heating system (based on original FORTRAN code from colleagues in mechanical engineering). Since this is a heat transfer problem, the initial theory is the general form of the conservation of thermal energy. This general form is a partial differential equation in 3D space and time [3]. To specialize the theory to the solar water heating tank, we introduced the following assumptions:

- The water in the tank is fully mixed.
- The density of water has no spatial variation.
- The specific heat capacity of water has no spatial variation.

The refined theory is an ordinary differential equation that depends only on time. This new theory is parameterized by material properties and the area and thermal flux between two adjacent bodies. This theory in turn is specialized into two more theories: one where the adjacent bodies are the tank and the heating coil, and one where the bodies are the water and a phase change material.

By systematically refining the theory, and generating documentation, human beings can validate the assumptions and the mathematical model. By capturing the knowledge, changes can be made to realize a family of mathematical models. Moreover, the knowledge can be reused for other heat transfer problems.

Step 5 : Code-level Choices

We can choose languages, how "modular" the generated code is, whether we want programs or libraries, the level of logging and comments, etc.

```
code :: CodeSpec
code = codeSpec fullSI choices allMods

choices :: Choices
choices = defaultChoices {
 lang = [Python, Cpp, CSharp, Java, Swift],
 modularity = Modular Separated,
 impType = Program, logFile = "log.txt",
 logging = [LogVar, LogFunc],
 comments = [CommentFunc, CommentClass, CommentMod],
 doxVerbosity = Quiet,
 dates = Hide,
 onSfwrConstraint = Exception, onPhysConstraint = Exception,
 inputStructure = Bundled,
 constStructure = Inline, constRepr = Const
}
```

Step 6

We can now generate multiple artifacts (here: specification in HTML and LaTeX, code in multiple languages with choices, dependency diagram and log of all choices used) as desired:

```
main :: IO()
main = do
  setLocaleEncoding utf8
  gen (DocSpec (docChoices SRS [HTML, TeX]) "GlassBR_SRS") srs
       printSetting
  genCode choices code
  genDot fullSI
  genLog fullSI printSetting
```

4 CONCLUDING REMARKS

For well understood domains, building software ought to be a matter of engineering, based on solid scientific foundations. The ultimate test of "well understood" is being able to teach the domain language to a computer. We have shown samples of a process and implementation framework for generating all of the software artifacts for (well understood) software, from the natural knowledge base of the domain.

[3]with some additional abbreviations for space.

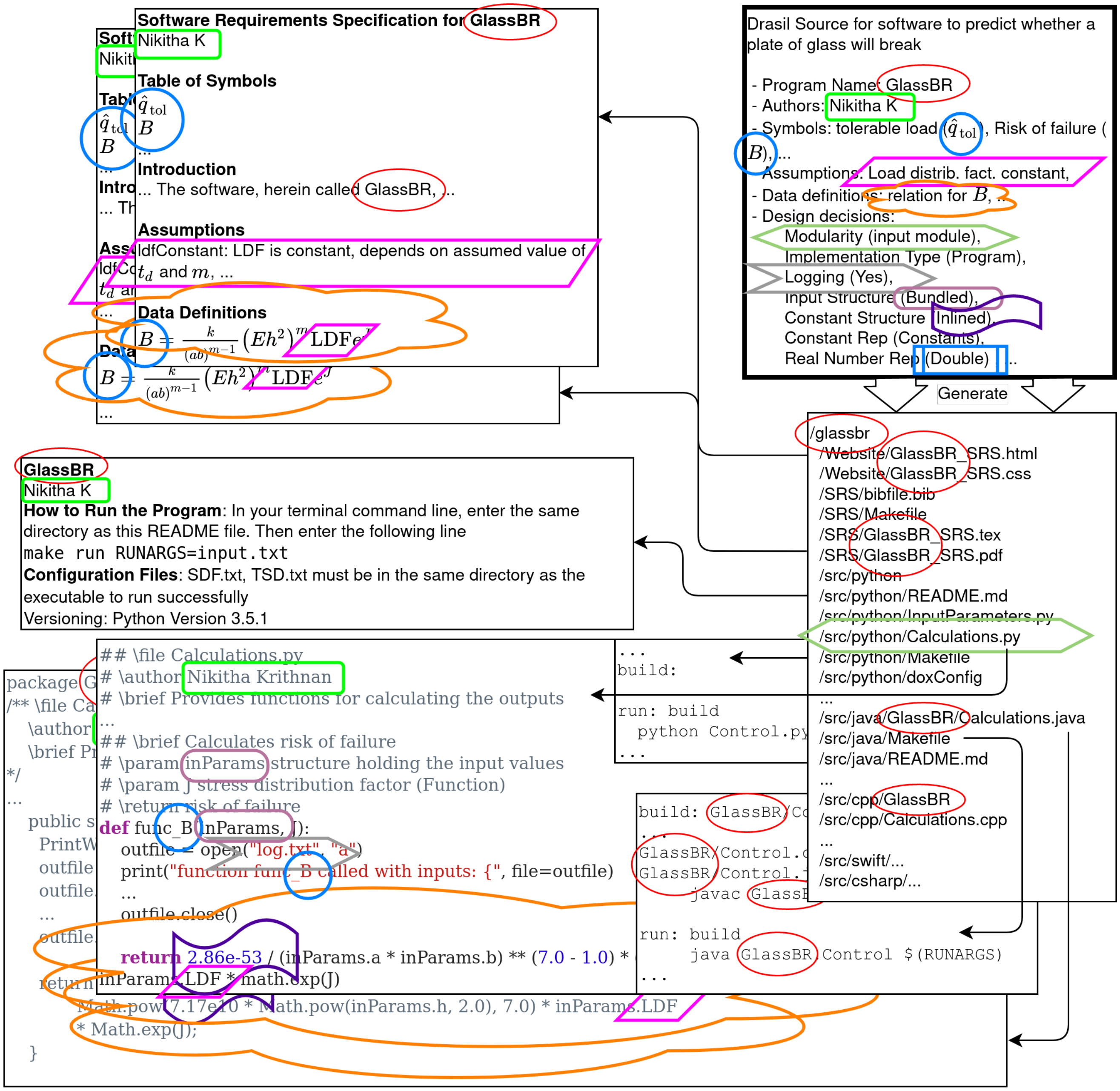

Figure 1: Colors and shapes mapping from captured knowledge to generated artifacts.

We take advantage of inherent knowledge duplication between artifacts in a project, and between projects, by capturing the knowledge once and providing means to transform that information into all of the views needed within a given project. Developers can realize long-term productivity with documentation and code that are consistent by construction.

Codifying scientific, engineering and computational knowledge is challenging, but success will completely transform the development of software, and software families, in well understood domains. Our process will remove human errors from generating and maintaining documentation, code, test cases and build environments, since the mundane details are handled by the generator. With the new software tools, we can potentially detect inconsistencies between theory via inter-theory consistency constraints. Moreover, we can explicitly track the ramifications of a proposed change. With the right up-front investment, we can have sustainable software because stable knowledge is separated from rapidly changing assumptions and design decisions.

REFERENCES

[1] ASTM. 2009. Standard Practice for Determining Load Resistance of Glass in Buildings.

[2] W. Lynn Beason, Terry L. Kohutek, and Joseph M. Bracci. 1998. Basis for ASTME E 1300 Annealed Glass Thickness Selection Charts. *Journal of Structural Engineering* 124, 2 (February 1998), 215–221.

[3] R.B. Bird, E.N. Lightfoot, and W.E. Stewart. 2002. *Transport Phenomena*. J. Wiley. https://books.google.ca/books?id=wYnRQwAACAAJ

[4] Barry W. Boehm. 1987. Improving Software Productivity. *Computer* (1987), 43–47.

[5] Kimiz Dalkir. 2011. *Knowledge Management in Theory and Practice*. The MIT Press.

[6] William M. Farmer. 2007. Biform Theories in Chiron. In *Towards Mechanized Mathematical Assistants*, Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 66–79.

[7] IEEE. 1998. Recommended Practice for Software Requirements Specifications. *IEEE Std 830-1998* (Oct. 1998), 1–40. https://doi.org/10.1109/IEEESTD.1998.88286

[8] 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.

[9] Suzanne Robertson and James Robertson. 1999. *Mastering the Requirements Process*. ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, Chapter Volere Requirements Specification Template, 353–391.

[10] W. Spencer Smith, Lei Lai, and Ridha Khedri. 2007. Requirements Analysis for Engineering Computation: A Systematic Approach for Improving Software Reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation* 13, 1 (Feb. 2007), 83–107. https://doi.org/10.1007/s11155-006-9020-7