When Capturing Knowledge Improves Productivity

Anonymous Author(s)

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:
Anonymous Author(s). 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, 4 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.

Pittsburgh '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

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, where inputs are labour, but adjust the outputs to be knowledge and user satisfaction, where user satisfaction acts as a proxy for effective quality (see [4] for more). 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.
(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* [2]. 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.

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 [3]) in such a scenario?

(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,
   (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 in the problem domain, as well as their associated constraints.
(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.

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 to enable it. This is usually tacit knowledge that entirely resides in developers' heads.

[1]A current meta-design choice.

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* [**?**].

3 AN EXAMPLE

We have built the needed infrastructure. It consists of 60kloc of Haskell implementing a series of interaction Domain Specific Languages (DSLs) for knowledge encodings, mathematical expressions, theories, English fragments, code generation and document generation[2] Instead, we will focus on an example that illustrates the ideal process, including how we capture information and the potential artifacts we can generate. To make the examples grounded, we'll focus mainly on the code and software artifacts generated for one of our examples: GlassBR, which is software used to predict the blast risk involved for a glass window. The requirements for the software are based on an American Standard Test Method (ASTM) standard [1].

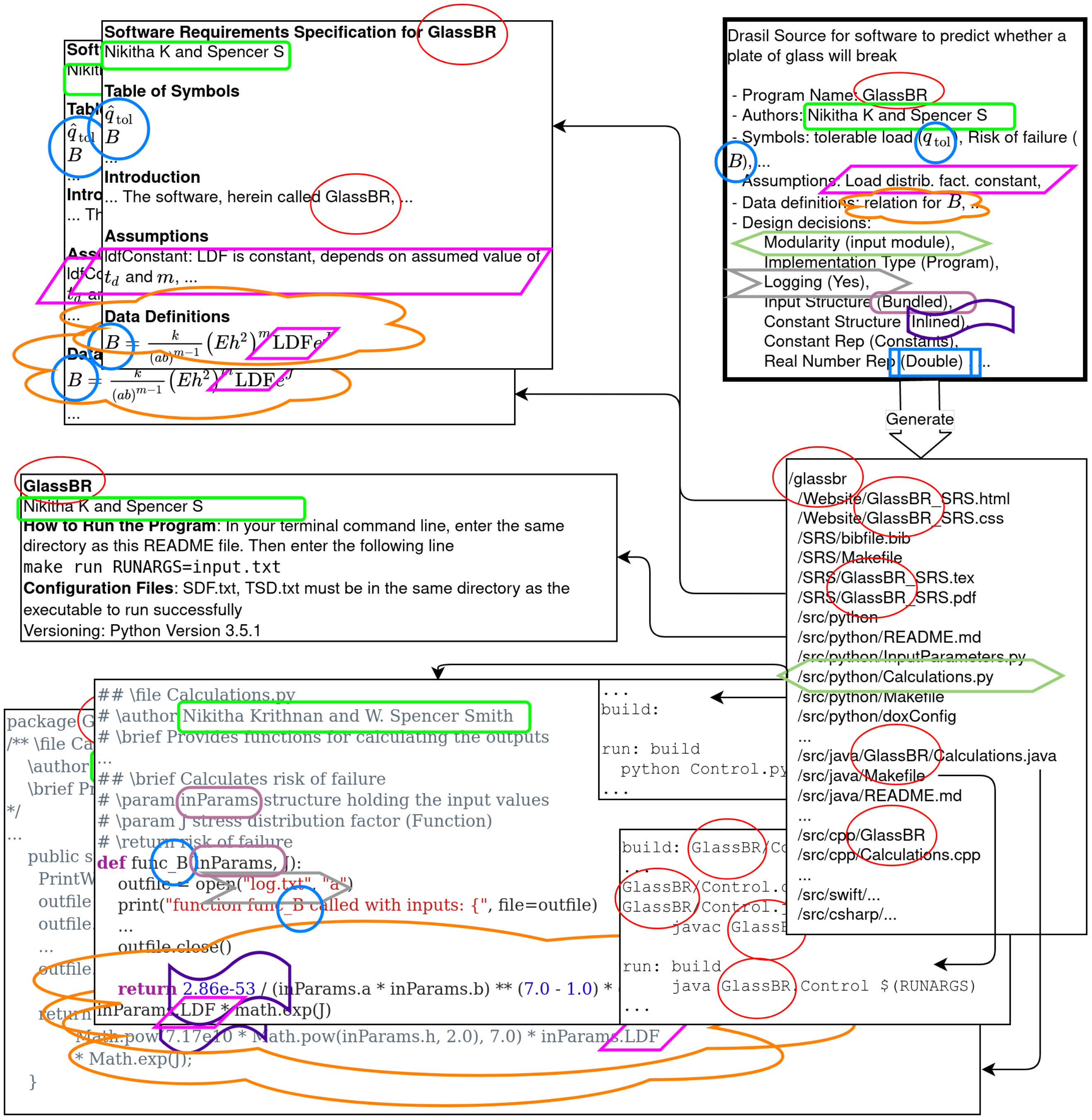
Figure 1 shows, for the GlassBR example, the transformation of captured knowledge (as shown in the darker bordered box at the top right). The generated artifacts come from the weaving together of information, as outlined in Step 6 of the ideal process. An example of this weaving can be seen in the name of the software. In the generated artifacts the name GlassBR appears more than 80 times — in the folder structure, in the requirements specification, in the README file, in the Makefile, and in the source code. In a conventional software project changing the name across all artifacts is surprisingly difficult — difficult enough that the change may not be made. With our approach a change like this is made by one modification to the source knowledge and regeneration. By capturing domain knowledge, we facilitates more than just renaming. For instance, if the assumption of a constant Load Distribution Factor (LDF) changes, the regenerated software will have LDF as an input variable. We also capture design decisions, like 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.

Step 3a: Base Knowledge

```
fullyT , glassTypeFac , heatS , iGlass , lGlass :: CI
fullyT = commonIdeaWithDict "fullyT" (nounPhraseSP "fully tempered") "FT"
      [idglass]
glassTypeFac = commonIdeaWithDict "glassTypeFac" (nounPhraseSP "glass type
      factor") "GTF" [idglass]
heatS = commonIdeaWithDict "heatS" (nounPhraseSP "heat strengthened")
      "HS"[idglass]
iGlass = commonIdeaWithDict "iGlass" (nounPhraseSP "insulating glass")
      "IG"[idglass]
lGlass = commonIdeaWithDict "lGlass" (nounPhraseSP "laminated glass") "LG"
      [idglass]


risk :: DataDefinition
risk = ddE riskQD
  [dRef astm2009, dRefInfo beasonEtAl1998 $ Equation [4, 5],
  dRefInfo campidelli $ Equation [14]]
  Nothing "riskFun" [aGrtrThanB, hRef, ldfRef, jRef]
  where riskQD = mkQuantDef riskFun riskEq
```

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



Drasil Source for software to predict whether a plate of glass will break

- Program Name: GlassBR
- Authors: Nikitha K and Spencer S
- Symbols: tolerable load ($\hat{q}_{\text{tol}}$), Risk of failure (B), ...
- Assumptions: Load distrib. fact. constant,
- Data definitions: relation for B, ...
- Design decisions:
  Modularity (input module),
  Implementation Type (Program),
  Logging (Yes),
  Input Structure (Bundled),
  Constant Structure (Inlined),
  Constant Rep (Constants),
  Real Number Rep (Double), ...

Software Requirements Specification for GlassBR

Nikitha K and Spencer S

Table of Symbols

$\hat{q}_{\text{tol}}$

B

...

Introduction

... The software, herein called GlassBR, ...

Assumptions

ldfConstant: LDF is constant, depends on assumed value of t_d and m, ...

Data Definitions

$B = \frac{k}{(ab)^{m-1}} (Eh^2)^m \text{LDF} e^J$

...

```
/glassbr
  /Website/GlassBR_SRS.html
  /Website/GlassBR_SRS.css
  /SRS/bibfile.bib
  /SRS/Makefile
  /SRS/GlassBR_SRS.tex
  /SRS/GlassBR_SRS.pdf
  /src/python
  /src/python/README.md
  /src/python/InputParameters.py
  /src/python/Calculations.py
  /src/python/Makefile
  /src/python/doxConfig
  ...
  /src/java/GlassBR/Calculations.java
  /src/java/Makefile
  /src/java/README.md
  ...
  /src/cpp/GlassBR
  /src/cpp/Calculations.cpp
  ...
  /src/swift/...
  /src/csharp/...
```

GlassBR
Nikitha K and Spencer S
How to Run the Program: In your terminal command line, enter the same directory as this README file. Then enter the following line
`make run RUNARGS=input.txt`
Configuration Files: SDF.txt, TSD.txt must be in the same directory as the executable to run successfully
Versioning: Python Version 3.5.1

```
## \file Calculations.py
# \author Nikitha Krithnan and W. Spencer Smith
# \brief Provides functions for calculating the outputs
...
## \brief Calculates risk of failure
# \param inParams structure holding the input values
# \param J stress distribution factor (Function)
# \return risk of failure
def func_B(inParams, J):
    outfile = open("log.txt", "a")
    print("function func_B called with inputs: {", file=outfile)
    ...
    outfile.close()

    return 2.86e-53 / (inParams.a * inParams.b) ** (7.0 - 1.0) *
inParams.LDF * math.exp(J)
```

```
...
build:

run: build
  python Control.py
...
```

```
build: GlassBR/Co
...
GlassBR/Control.c
GlassBR/Control.
    javac GlassB

run: build
    java GlassBR.Control $(RUNARGS)
...
```

```
package G
/** \file Ca
  \author
  \brief Pr
*/
...
  public s
    PrintW
    outfile
    outfile
    ...
    outfile
  return
    Math.pow(7.17e10 * Math.pow(inParams.h, 2.0), 7.0) * inParams.LDF
    * Math.exp(J);
  }
```

Figure 1: Mapping between changes in DSL source code to the generated artifacts. The different colors and shapes show the connection between the source knowledge and the generated artifacts.

```
riskEq = (sy sflawParamK $/
  (mulRe (sy plateLen) (sy plateWidth) $^ (sy sflawParamM $- exactDbl
        1))) `mulRe`
  ((sy modElas `mulRe` square (sy minThick)) $^ sy sflawParamM)
        `mulRe` sy lDurFac `mulRe` exp (sy stressDistFac)
```

Step 3b: Coherent Narrative

As per the requirements template for scientific software [5, 6], we have a goal: "**Predict-Glass-Withstands-Explosion**: 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." The goal statement is generated via:

```
willBreakGS :: ConceptInstance
willBreakGS = cic "willBreakGS" (foldlSent [S "Analyze" `S.and_`
S "predict whether the", phrase glaSlab, S "under consideration will be able",
S "to withstand the", phrase explosion `S.of_` S "a certain", phrase degree_',
S "which is calculated based on", phrase userInput])
"Predict-Glass-Withstands-Explosion" goalStmtDom
```

Step 4a: Specialization of Theories

Theories are specialized via assumptions, like this one: "**glassCondition**: Following astm2009 (pg. 1), this practice does not apply to any form of wired, patterned, etched, sandblasted, drilled, notched, or grooved glass with surface and edge treatments that alter the glass strength. (RefBy: UC:Accommodate-Altered-Glass.)" This assumptions is generated via:

```
glassConditionDesc :: Sentence
glassConditionDesc = foldlSent [S "Following", complexRef astm2009 (Page [1])
      `sC`
  S "this", phrase practice, S "does not apply to any form of", foldlList Comma
        Options $ map S ["wired",
  "patterned", "etched", "sandblasted", "drilled", "notched", "grooved glass"],
        S "with",
  phrase surface `S.and_` S "edge treatments that alter the glass strength"]
```

Step 4b: ?

We create a closed harmonic system containing all related knowledge (in particular, the grounded theories), which is as simple as collecting them into a single system.

```
iMods :: [InstanceModel]
iMods = [pbIsSafe, lrIsSafe]

si :: SystemInformation
si = SI {
_sys = glassBR, _kind = Doc.srs, _authors = [nikitha, spencerSmith],
_purpose = purpDoc glassBR Verbose, _quants = symbolsForTable,
_concepts = [] :: [DefinedQuantityDict], _instModels = iMods,
_datadefs = GB.dataDefs, _configFiles = configFp,
_inputs = inputs, _outputs = outputs,
_defSequence = qDefns, _constraints = constrained,
_constants = constants, _sysinfodb = symbMap,
_usedinfodb = usedDB, refdb = refDB
}
```

Step 5

We make choices about the software representations of the theories:

```
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,
  auxFiles = [SampleInput "../../datafiles/glassbr/sampleInput.txt", ReadME]
}
```

Step 6

A final executable program which should take the knowledge discussed above, and use pre-made printers/generators to generate software artifacts, SRS documents, etc:

```
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.

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] 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.

[2] 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.

[3] 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.

[4] Spencer Smith and Jacques Carette. 2020. Long-term Productivity for Long-term Impact, arXiv report. https://arxiv.org/abs/2009.14015. arXiv:2009.14015 [cs.SE]

[5] W. Spencer Smith and Lei Lai. 2005. A New Requirements Template for Scientific Computing. In *Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP'05*, J. Ralyté, P. Ågerfalk, and N. Kraiem (Eds.). In conjunction with 13th IEEE International Requirements Engineering Conference, Paris, France, 107–121.

[6] 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