# THE DRASIL FRAMEWORK

# SUCCINCTLY VERBOSE: THE DRASIL FRAMEWORK

BY

DANIEL M. SZYMCZAK, M.A.Sc., B.Eng

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy(2021)                                    McMaster University

Computing & Software                                 Hamilton, Ontario, Canada



TITLE:                    Succinctly Verbose: The Drasil Framework


AUTHOR:                   Daniel M. Szymczak

                          M.A.Sc. (Software Engineering)

                          McMaster University

                          Hamilton, Ontario, Canada


SUPERVISORS:              Jacques Carette and Spencer Smith


NUMBER OF PAGES:    xiii, 34

# Lay Abstract

A lay abstract of not more 150 words must be included explaining the key goals and contributions of the thesis in lay terms that is accessible to the general public.

# Abstract

Abstract here (no more than 300 words)

*Your Dedication*

*Optional second line*

# Acknowledgements

Acknowledgements go here.

# Contents

# List of Figures

# List of Tables

# Notation, Definitions, and Abbreviations

[TODO: Update this —DS]

## Notation

$A \leq B$          A is less than or equal to B

## Definitions

**Softifact**      A portmanteau of 'software' and 'artifact'. The term refers to any of the artifacts (documentation, code, test cases, build instructions, etc.) created during a software project's development.

## Abbreviations

**QA**          Quality Assurance

**SI**           Système International d'Unités

**SRS**        Software Requirements Specification

# Chapter 1

# Introduction

[Introduce the term softifact somewhere in here —DS]

[Pain points and problems come first, then solution is docs, then problems with writing docs, then the rest of this. —DS] Documentation is good(**?**), yet it is not often prioritized on software projects. Code and other software artifacts say the same thing, but to different audiences - if they didn't, they would be describing different systems.

Take, for example, a software requirements document. It is a human-readable abstraction of *what* the software is supposed to do. Whereas a design document is a human-readable version of *how* the software is supposed to fulfill its requirements. The source code itself is a computer-readable list of instructions combining *what* must be done and, in many languages, *how* that is to be accomplished.

[Put in figures of an example from GlassBR/Projectile here, showing SRS, DD, and code versions of the same knowledge]

[Figure] shows an example of the same information represented in several different views (requirements, detailed design, and source code). We aim to take advantage of

1

the inherent redundancy across these views to distill a single source of information, thus removing the need to manually duplicate information across software artifacts.

Manually writing and maintaining a full range of software artifacts (i.e. multiple documents for different audiences plus the source code) is redundant and tedious. Factor in deadlines, changing requirements, and other common issues faced during development and you have a perfect storm for inter-artifact syncronization issues.

How can we avoid having our artifacts fall out of sync with each other? Some would argue "just write code!" And that is exactly what a number of other approaches have tried. Documentation generators like Doxygen, Javadoc, Pandoc, and more take a code-centric view of the problem. Typically, they work by having natural-language descriptions and/or explanations written as specially delimited comments in the code which are later automatically compiled into a human-readable document.

While these approaches definitely have their place and can come in quite handy, they do not solve the underlying redundancy problem. The developers are still forced to manually write descriptions of systems in both code and comments. They also do not generate all software artifacts - commonly they are used to generate only API documentation targeted towards developers or user manuals.

We propose a new framework, Drasil, alongside a knowledge-centric view of software, to help take advantage of inherent redundancy, while avoiding manual duplication and synchronization problems. Our approach looks at what underlies the problems we solve using software and capturing that "common" or "core" knowledge. We then use that knowledge to generate our software artifacts, thus gaining the benefits inherent to the generation process: lack of manual duplication, one source to maintain, and 'free' traceability of information.

## 1.1   Value in the mundane

??

## 1.2   Scope

We are well aware of the ambitious nature of attempting to solve the problem of manual duplication and unnecessary redundancy across all possible software systems. Frankly, it would be highly impractical to attempt to solve such a broad spectrum of problems. Each software domain poses its own challenges, alongside specific benefits and drawbacks.

Our work on Drasil is most relevant to software that is well-understood and undergoes frequent change (maintenance). Good candidates for development using Drasil are long-lived (10+ years) software projects with artifacts of interest to multiple stakeholders. With that in mind, we have decided to focus on scientific computing (SC) software. Specifically, we are looking at software that follows the pattern 'input -¿ process -¿ output'.

SC software has a strong fundamental underpinning of well-understood concepts. It also has the benefit of seldomly changing, and when it does, existing models are not necessarily invalidated. For example, rigid-body problems in physics are well-understood and the underlying modeling equations are unlikely to change. However, should they change, the current models will likely remain as good approximations under a specific set of assumptions. For instance, who hasn't heard 'assume each body is a sphere' during a physics lecture?

## 1.3 Roadmap

## 1.4 Contributions & Publications

[After so much time working here, I think I've finally realized one of the true contributions of this thesis/Drasil. Not only the framework itself (which is still awesome), but also the process of breaking everything down and truly understanding softifacts at a deep level to operationalize our understanding of SE / system design in a way that makes all of this generation possible. With that in mind Drasil is just one means to that end. —DS]

[Minor point that came up in conversation: Parnas' paper on how/why to fake rational design. Our tool lets people fake it. It's all about change, there's no perfect understanding at the beginning and we need to change things on as we go. Drasil allows us to change everything to fake the rational design process at every step along the way. —DS]

[Continuous integration / refactoring are nothing new, but the way we used them ensured we were always at a steady-state where everything worked. —DS]

[Note that some of the code may not have been written by me directly, but was developed by the Drasil team —DS]

# Chapter 2

# Background

## 2.1 Software Artifacts

- (Reference a lot of Parnas)

## 2.2 Software Reuse and Software Families

- Bring up GNU/Linux and different distros as examples of software families - (Raspbian v Raspbian lite) = Debian--, etc.

## 2.3 Literate Programming

- Knuth - lots of Knuth - Gentleman and Lang 2012 - Johnson and Johnson 1997 - Literate programming using noweb - VNODE-LP - Physically Based Rendering ... - Al-Maati and Boujarwah 2002

## 2.4 Generative Programming

- ?

# Chapter 3

# A look under the hood:

# Our process

[Make sure we talk about continuous integration / git processes / etc —DS]

The first step in removing unnecessary redundancy is identifying exactly what that redundancy is and where it exists. To that end we need to understand what each of our software artifacts is attempting to communicate, who their audience is, and what information can be considered boilerplate versus system-specific. Luckily, we have an excellent starting point thanks to the work of many smart people - artifact templates.

Lots of work [cite some people who did this —DS] has been done to specify exactly what should be documented in a given artifact in an effort for standardization. Ironically, this has led to many different 'standardized' templates. Through the examination of a number of different artifact templates, we have concluded they convey roughly the same overall information for a given artifact. Most differences are stylistic or related to content organization and naming conventions, as we will demonstrate in

the following sections.

Once we understand our artifacts, we take a practical, example-driven approach to identifying redundancy through the use of existing software system case studies. For each of these case studies, we start by examining the source code and existing software artifacts to understand exactly what problem they are trying to solve. From there, we attempt to distill the system-specific knowledge and generalize the boilerplate.

## 3.1 A (very) brief introduction to our case study systems

[**NOTE: ensure each artifact has a 'who' (audience), 'what' (problem being solved), and 'how' (specific-knowledge vs boilerplate) - this last one may not be necessary — DS]

To simplify the process of identifying redundancies and patterns, we have chosen several case studies developed using common artifact templates, specifically those used by Smith et al. [source(**?**) —DS] Also, as mentioned in [[SCOPE] —DS], we have chosen software systems that follow the '*input*' $\rightarrow$ '*process*' $\rightarrow$ '*output*' pattern. These systems cover a variety of use cases, to help avoid over-specializing into one particular system-type.

The majority of the aforementioned case studies were developed to solve real problems, with a few minor exceptions listed below their relevant cards. [**NOTE: trying to find a good way to say 'these are not just things we cooked up to make Drasil look good' —DS] The cards below are meant to be used as a high-level reference to each case study, providing the general details at a glance. For the specifics of each

system, all relevant case study artifacts can be found at [Add a link here or put in appendices? —DS].

---

**Name**: **GlassBR**

**Problem being solved**: We need to efficiently and correctly predict whether a glass slab can withstand a blast under given conditions.

Relevant artifacts: [TODO —DS]

---

**Name**: **SWHS**

**Problem being solved**: Solar water heating systems incorporating phase change material (PCM) use a renewable energy source and provide a novel way of storing energy. A system is needed to investigate the effect of employing PCM within a solar water heating tank.

Relevant artifacts: [TODO —DS]

---

**Name**: **NoPCM**

**Problem being solved**: Solar water heating systems provide a novel way of heating water and storing renewable energy. A system is needed to investigate the heating of water within a solar water heating tank.

Relevant artifacts: [TODO —DS]

---

The NoPCM case study was created as a software family member for the SWHS case study. It was manually written, removing all references to PCM and thus re-modeling the system.

| |
|---|
| **Name**: **SSP** |
| **Problem being solved**: A slope of geological mass, composed of soil and rock and sometimes water, is subject to the influence of gravity on the mass. This can cause instability in the form of soil or rock movement which can be hazardous. A system is needed to evaluate the factor of safety of a slope's slip surface and identify the critical slip surface of the slope, as well as the interslice normal force and shear force along the critical slip surface.<br><br>Relevant artifacts: [TODO —DS] |

| |
|---|
| **Name**: **Projectile** |
| **Problem being solved**: A system is needed to efficiently and correctly predict the landing position of a projectile.<br><br>Relevant artifacts: [TODO —DS] |

The Projectile case study, was the first example of a system created solely in Drasil, i.e. we did not have a manually created version to compare and contrast with through development. As such, it will not be referenced often until [DRASILSECTION — DS] since it did not inform Drasil's design or development until much further in our process. The Projectile case study was created post-facto to provide a simple,

understandable example for a general audience as it requires, at most, a high-school level understanding of physics.

| |
|---|
| **Name**: **GamePhysics** |
| **Problem being solved**: Many video games need physics libraries that simulate objects acting under various physical conditions, while simultaneously being fast and efficient enough to work in soft real-time during the game. Developing a physics library from scratch takes a long period of time and is very costly, presenting barriers of entry which make it difficult for game developers to include physics in their products.<br><br>Relevant artifacts: [TODO —DS] |

After carefully selecting our case studies, we went about a practical approach to find and remove redundancies. The first step was to break down each artifact type and understand exactly what they are trying to convey.

## 3.2   Breaking down softifacts

As noted earlier, for our approach to work we must understand exactly what each of our artifacts are trying to say and to whom. By selecting our case studies from those developed using common artifact templates, we have given ourselves a head start on that process, however, there is still much work to be done.

To start, we look at the Software Requirements Specification (SRS). The SRS (or some incarnation of it) is one of the most important artifacts for any software

*Figure showing the ToC of Smith et al. template*

Figure 3.1: The Table of Contents from the Smith et al. template

*Figure showing the Ref Section of one case study, split into multiple subfigures - case study TBD*

Figure 3.2: The reference sections of [TBD —DS] respectively

project as it specifies what problem the software is trying to solve. There are many ways to state this problem, and the template from Smith et al. has given us a strong starting point. Figure 3.1 shows the table of contents for an SRS using the Smith et al. template.

With the structure of the document in mind, let us look at several of our case studies' SRS documents to get a deeper understanding of what each section truly represents. Figure 3.2 shows the reference section of the SRS for [TODO —DS]. Each of the case studies' SRS contains a similar section so for brevity we will omit the others here, but they can be found at [TODO —DS]. We will look into the case studies in more detail later, for now we will try to ignore the superficial differences in each of them while we look for commonality. ~~We are strictly looking for patterns! Patterns will give us insight into the root of *what* is being said in each section.~~[Not really true, will need to rework —DS]

Looking at the Table of Symbols, Table of Units, and Table of Abbreviations and Acronyms from Figure 3.2 we can see that, barring the table values themselves, they are almost identical. The Table of Symbols is simply a table of values, akin to a glossary, specific to the symbols that appear throughout the rest of the document. For each of those symbols, we see the symbol itself, a brief description of what that

symbol represents, and the units it is measured in, if applicable. Similarly, the Table of Units lists the Système International d'Unités (SI) Units used throughout the document, their descriptions, and the SI name. Finally, the table of Abbreviations and Acronyms lists the abbreviations and their full forms, which are essentially the symbols and their descriptions for each of the abbreviations.

While the reference material section should be fairly self-explanatory as to what it contains, other sections and subsections may not be so clear from their name alone. For example, it may not be clear offhand of what constitutes a theoretical model compared to a data definition or an instance model. One may argue that the author of the SRS, particularly if they chose to use the Smith et al. template, would need to understand that difference. However, it is not clear whether the intended audience would also have such an understanding. Who is that audience? We will explore that in Section 3.3.

Returning to our exercise of breaking down each section of the SRS to determine the subtleties of *what* is contained therein (the details are omitted for brevity, although the overall process is very much akin to that of our breakdown of the Reference Material section) it should be unsurprising that each section maps to the definition provided in the Smith et al. template. However, as noted above, we can see distinct differences in the types of information contained in each section. Again we find some is boilerplate text meant to give a generic (non-specific) overview, some is specific to the proposed system, and some is in-between: it is specific to the problem domain for the proposed system, but not necessarily specific to the system itself.

## 3.3   Softifact Summary

Parnas (**?**) does an excellent job of defining the target audience for each of the most common softifacts and we extend that alongside our work. A summary can be found in Table 3.1.

Table 3.1: A summary of the Audience for each of the most common softifacts and what problem that softifact is solving

| Softifact | Who (Audience) | What (Problem) |
|---|---|---|
| SRS | 1. Software Architects<br><br>2. QA analysts | Define exactly what specification the software system must adhere to. |
| Module Guide | 1. All developers<br><br>2. QA analysts | [TODO —DS] |
| Module Interface Specifications | 1. Developers who implement the module<br><br>2. Developers who use the module<br><br>3. QA analysts | [TODO —DS] |
| Verification and Validation Plan | 1. Developers who implement the module<br><br>2. QA analysts | [TODO —DS] |

## 3.4   Patterns and repetition and patterns and repetition – (OR – Repeating patterns and patterns that repeat –)

[**Make sure to mention "We actually found that some info is boilerplate, some is system-specific, and some is general to all members of a software family, but more specific than generic boilerplate" —DS]

From the above sections, we see many emerging patterns in our softifacts. Ignoring, for now, the organizational patterns from the Smith et al. templates we can already see simple patterns emerging.

Returning to our example from Section 3.2, looking only at the reference section of our SRS template, we have already found three subsections that contain the majority of their information in the same organizational structure: a table of symbols and general information relevant to those symbols. Additionally, we can see that the Table of Units and Table of Symbols have an introductory blurb preceding the tables themselves, whereas the Table of Abbreviations and Acronyms does not. Inspecting across case studies, we can see that the introduction to the Table of Units is simply boilerplate text dropped into each case study verbatim, thus is completely generic and applicable to *any* software system using SI units. The introduction to the Table of Symbols appears to be boilerplate across several examples, however, it does have some variation (see: GamePhysics compared to GlassBR). These variations reveal the obvious: the variability between systems is greater than simply a difference in choice of symbols, and so there is some system-specific knowledge being encoded. We can intuit this conclusion based solely on each system solving a different problem,

however, we have confirmed this by solely looking at the structure (patterns) of one section of the SRS.

The reference section of the SRS provides a lot of knowledge in a very straightforward and organized manner. The basic units provided in the table of units give a prime example of fundamental, global knowledge shared across domains. Nearly any system involving physical quantities will use some of these units. On the other hand, the table of symbols provides system/problem-domain specific knowledge that will not be useful across unrelated domains. For example, the stress distribution factor $J$ from GlassBR may appear in several related problems, but would be unlikely to be seen in something like SWHS, NoPCM, or Projectile. Finally, acronyms are very context-dependent. They are often specific to a given domain and, without a coinciding definition, it can be very difficult for even the target audience to understand what they refer to. Within one domain, there may be several acronyms meaning different things, for example: PM can refer to Product Manager, Project Manager, Program Manager, etc.

By continuing to breakdown the SRS and other softifacts, we are able to find many more patterns. For example, we see the same concept being introduced in multiple areas within a single artifact and across artifacts in a project. [Example from one of the figures in the previous section. Preferably something like a DD or TM that shows up within a single doc multiple times]. We also see patterns of commonality across software family members (The SWHS and NoPCM case studies) as they have been developed to solve similar, or in our case nearly identical, problems.

- inter-project (repetition throughout different views + other patterns.) vs intra-project knowledge (repetition across projects/family members, minor modifications,

but fundamentally the same + other patterns.) - Hint at chunkifying/parceling out the fundamental (system/view-agnostic) knowledge vs the specific knowledge

## 3.5   Organizing knowledge - a fluid approach

**Subsec roadmap: - We see the patterns above, we can generalize a lot of that - Direct repetition (copy-paste) vs indirect repetition (view-changes) require us to pull together knowledge from all artifacts into one place - Some can be derived automatically, the rest must be explicitly stated - We need to create a categorization system (hint at chunks) that is both robust and extensible to cover a wide variety of use cases. - Finally the templates give us structure

   **NOTE: Under the hood section should explain the process of how we determined what we needed to do. What we ended up doing should come in the following section(s) - no 'real' implementation details, only conceptual stuff here.

## 3.6   The seeds of Drasil

**Subsec roadmap: – Summarize the above subsections and lead into next section – Add relevant information that doesn't quite fit above and isn't implementation related – 'Relevant buckshot section'

# Chapter 4

# Drasil

In this chapter we will introduce the Drasil framework and some details of its implementation including knowledge-capture mechanisms, the domain specific languages used throughout, and how all of these pieces are brought together in a human-usable way to generate softifacts.

## 4.1 What Drasil is and isn't

* Basically just restating some things from the intro in more depth - Not a silver bullet - Built around a specific set of assumptions, for a particular class of problems - NOT an ontology / ontology builder

Manually writing and maintaining a full range of softifacts is redundant, tedious, and often softifacts fall our of sync with each other. Drasil is a framework built to tackle these problems.

Contrary to documentation generators like Doxygen, Javadoc, and Pandoc which take a code-centric view of the problem and rely on manual redundancy – i.e. natural-language explanations written as specially delimited comments which can then be weaved into API documentation alongside code – Drasil takes a knowledge-centric, redundancy-limiting, fully traceable single source approach to generating all softifacts.

However, Drasil is not a panacea for all the woes of software development. Even the seemingly well-defined issues of unnecessary redundancy and manual duplication turn out to be large, many-headed beasts existing across a multitude of software domains; each with their own benefits, drawbacks, and challenges.

To reiterate: Drasil has not been designed as a silver-bullet. It is a specialized tool meant to be used in well-understood domains for software that will undergo frequent maintenance and/or changes. In deciding whether Drasil would be useful for developing software to tackle a given problem, we recommend identifying those projects that are long-lived (10+ years) with softifacts relevant to multiple stakeholders. For our purposes, as mentioned earlier, we have focused on SC software that follows the input $\rightarrow$ process $\rightarrow$ output pattern. SC software has the benefit of being relatively slow to change, so models used today may not be updated or invalidated for some time, if ever. Should that happen, the models will likely still be applicable given a set of assumptions or assuming certain margin for error.

With Drasil being built around this specific class of problems, we remain aware that there are likely many in-built assumptions that could affect its applicability to

other domains in its current state. Expanding Drasil's reach is an avenue for future work.

The Drasil framework relies on a knowledge-centric approach to software specification and development. We attempt to codify the foundational theory behind the problems we are attempting to solve and operationalize it through the use of generative technologies. By doing so, we can reuse common knowledge across projects and maintain a single source of truth in our knowledge database.

Given how important knowledge is to Drasil, one might think we are building ontologies or ontology generators. This is *not* the case. We are not attempting to create a source for all knowledge and relationships inside a given field. We are merely using the information we have available to build up knowledge as needed to solve problems. Over time, this may take on the appearance of an ontology, but Drasil does not currently enforce any strict rules on how knowledge should be captured, outside of its type system and some best practice recommendations. We will explore knowledge capture in more depth in Section 4.2.

## 4.2 Our Ingredients: Organizing and Capturing Knowledge

- Organization of knowledge implies a need for knowledge-capture mechanisms at different levels (different levels of abstraction / specificity) - segues right into chunks/hierarchy - project-specific vs DB of knowledge - Make it clear this is NOT an ontology - Look for interesting examples (synonyms, acronyms, complexity) from case studies and refer to them here (again if was covered in Organizing Knowledge)

## 4.3    Recipes: Codifying Structure

- Organized knowledge is fine, but is essentially just a collection of (collections of) definitions. Pretty meaningless on its own so we need the structure (in our case from the templates / case studies) to have meaning. - Each softifact has its own recipe for combining knowledge - As we consider softifacts "views" of the knowledge, we need to combine/transform/manipulate the knowledge into a meaningful form for the given view - ex: Math formula for human-readable doc, Function/method for code (show examples). - Recipes define the "how" and "where" of putting together the knowledge. The rendering engine reads the recipe and follows its instructions.

## 4.4    Cooking it all up: Generation/Rendering

- Recipes are little programs - Each recipe can be rendered a number of ways, based on parameters fed to the generator. - Implicit parameters vs explicit: Ex. an SRS will always be rendered based on the recipe, but its output will either be LaTeX or HTML based on an explicit choice. Implicit params fed to gen table of symbols/A&A / ToU.

## 4.5    Reimplementing the case studies in Drasil

[TODO: move this / rearrange it into a "Iteration and refinement" section that doesn't discuss the specific results with the case studies. —DS] - Practical approach to iron out kinks / find holes in Drasil - Find places to improve upon the existing case studies - update as you go mindset - Observe the amount of effort required to correct errors

- show examples - Most of the code well show off should be in here.

# Chapter 5

# Results

In this chapter we will discuss our observations following the reimplementation of our case studies using Drasil. At present these observations are anecdotal in nature as we have not yet been afforded the time to design more rigorous experiments for data collection due to Drasil being very much in flux and undergoing constant development. We will discuss more about experimental data collection in "Future Work"(Chapter 6).

## 5.1   "Pervasive" Bugs

One of the first, and most curious, observations made while using Drasil was that of so called *pervasive bugs*. While we usually consider bugs to be something we wish to avoid at all costs, this is a case where the pervasiveness of bugs themselves is beneficial. Since we are generating *all* our softifacts from a single source, a bug in that source will result in a bug occurring through *every single softifact*. The major consequence is that the bug now has increased visibility, so is more likely to be discovered.

[Need a salient example of a pervasive bug we found here, or description of a handl of "we found these along the way just because they were so readily visible" —DS]

Pervasive bugs have another unique selling point. Consider a piece of software developed using generally accepted processes like waterfall or agile. After the initial implementation is complete, any bugs found are typically fixed by updating the code and other pertinent softifacts. As mentioned earlier, there are many instances, especially those involving tight deadlines or where non-executable softifacts are not prioritized, where the softifacts can fall out of sync with the implemented solution. As such we may end up with inconsistent softifacts that are wrong in (potentially) different ways. The following example involves the equation for Non-Factored Load (NFL) taken from the GlassBR case study:

$$NFL = \frac{\hat{q}_{\text{tol}} E h^4}{(ab)^2}$$

and its code representation (in python):

```python
def func_NFL(inParams, q_hat_tol):
    return q_hat_tol * 7.17e10 * inParams.h ** 4.0 / (inParams.a * inParams.b) ** 2.0
```

At a glance, are the code and formula equivalent?

It is difficult to say without confirming the value of $E$ is defined as $71.7 \cdot 10^9$, which is equivalent to the value used in the python code. However, if both softifacts were generated using Drasil then we have added confidence due to them being generated from the same source. Now if we determine there is a bug, we can look at either the formula or implementation – whichever we are more comfortable debugging – to determine how the source should be updated. As such, pervasive bugs give us peace of mind that our softifacts are consistent, even in the face of bugs.

## 5.2   Originals vs. Reimplementations

Link to original case studies and reimplementations Highlight key (important) changes with explanations Show off major errors/oversights Original softifacts LoC vs number of Drasil LoC to reimplement (and compare the output of Drasil - multiple languages, etc.)

## 5.3   Design for change

[GlassBR /1000 example —DS]

Designing for change can be difficult, especially when dealing with software with a long (10+ year) expected lifespan. Through our use of Drasil in updating the case studies, it has become obvious that Drasil expedites our ability to design for change.

Having only a single source to update accelerates implementation of desired changes, which we have demonstrated numerous times throughout Drasil's development and the reimplementation of our case studies. One salient example was in the GlassBR case study.

[Fill in the example details here —DS]

## 5.4   Usability

One of Drasil's biggest issues is that of usability. Unless one reads the source code or has a member of the Drasil team working with them, it can be incredibly difficult, or even impossible, to create a new piece of software in Drasil.

As seen in the examples from [SECTION], while the recipe language is fairly readable, the knowledge-capture mechanisms are arcane and determining which knowledge has already been added to the database can be very difficult. As our living knowledge-base expands, this will become even more difficult, particularly for those concepts with many possible names.

- As the above mentions, not great, but CS students / summer interns picked it up fast enough to make meaningful changes in a short time period.

# Chapter 6

# Future Work

[This can probably be a section at the end of results / conclusion instead? —DS]

Development of Drasil is ongoing and the framework is still being iterated upon to date. In this section we present areas we believe have room for improvement along with plans for additional features to be added in the long-term.

## 6.1   Typed Expression Language

## 6.2   Model types

## 6.3   Usability

As mentioned in the results section, usability remains a great area for improvement for Drasil. Work to create a visual front-end for the framework has been planned and we hope to eventually get to the point of usability being as simple as drag-and-drop or similar mechanisms.

Work on usability will address each of the core areas of development with Drasil: knowledge-capture of system-agnostic information, knowledge-capture of system-specific information, and recipe creation and modification.

## 6.4 Many more artifacts/more document knowledge

Journal papers, Jupyter notebooks, lesson plans, etc.

## 6.5 More display variabilities

## 6.6 More languages (?)

## 6.7 More scientific knowledge

Med, Chem, etc.

## 6.8 More computational knowledge

Higher order ODEs, linear system solvers, external libs, etc.

[More software engineering/process knowledge —DS]

## 6.9  Measuring Drasil's Effectiveness

We have made many observations as to how we believe Drasil can improve the lives of developers (Chapter 5), however, we have not yet backed that up with hard data. We need to design several experiments with differing goals to test Drasil's effectiveness in improving developer quality of life. Here we propose several experiments:

[TBD —DS]

# Chapter 7

# Conclusion

[Should Future work be collapsed into here? —DS]

Every thesis also needs a concluding chapter

# Appendix A

# Your Appendix

Your appendix goes here.

# Appendix B

# Long Tables

This appendix demonstrates the use of a long table that spans multiple pages.

| Col A | Col B | Col C | Col D |
|:-----:|:-----:|:-----:|:-----:|
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |

*Continued on the next page*

*Continued from previous page*

| Col A | Col B | Col C | Col D |
| :---: | :---: | :---: | :---: |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |
| A | B | C | D |

# Bibliography