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 (2022) McMaster University

(department of computing & software) Hamilton, Ontario, Canada

TITLE:	Succinctly Verbose: The Drasil Framework
AUTHOR:	Daniel M. Szymczak M.A.Sc.
SUPERVISOR:	Jacques Carette and Spencer Smith
NUMBER OF PAGES:	xiv, 74

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

Declaration of Academic Achievement

The student will declare his/her research contribution and, as appropriate, those of colleagues or other contributors to the contents of the thesis.

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

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* $\rightarrow$ *process* $\rightarrow$ *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?

SC software could also benefit from buy-in to good software development practices

as many SC software developers put the emphasis on science and not development [16]. Rather than following rigid, process-heavy approaches deemed unfavourable [4], developers of SC software choose to use knowledge acquisition driven [15], agile [37, 4, 1, 7], or amethododical [14] processes instead.

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

Software artifacts (or softifacts) come in a wide variety of forms and have existed since the first programs were created. In the broadest sense, we can think of softifacts as anything produced during the creation of a piece of software that serves some purpose. Any document detailing what the software should do, how it was designed, how it was implemented, how to test it, and so on would be considered a softifact, as would the source code whether as a text file, stack of punched cards, magnetic tapes, or other media.

Softifacts beyond just the source code are important to us for a number of reasons. Softifacts serve as a means of communication between different stakeholders involved in the software development process, or even different generations of developers involved in the production of a piece of software. They provide a common understanding of what the software is supposed to do, how it will be built, and how it will be tested. Softifacts also provide a record of the decisions that were made during

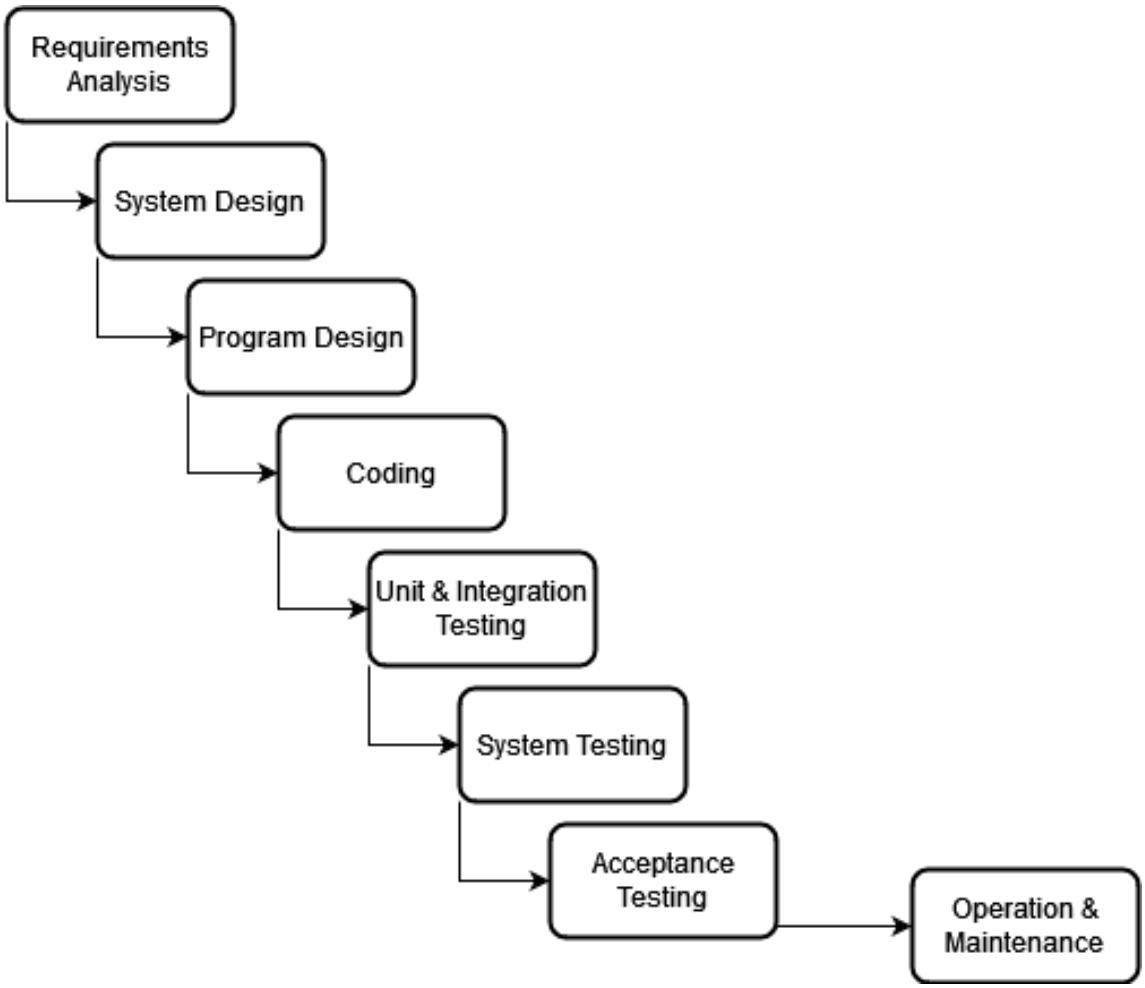

Figure 2.1: The Waterfall Model of Software Development

the development process which is important for future maintenance/modification of the software or for compliance reasons. Combined with the former, this gives us a means to verify and validate the software against its original requirements and design. The production, maintenance, and use of softifacts is largely dependent on the specific design process being used within a team.

Software design can follow a number of different design processes, each with their own collection of softifacts. A common, traditional approach is the Waterfall model

(Figure 2.1) of software development [31]. However, Parnas and Clements [30] detailed what they dubbed a "rational" design process; an idealized version of software development which includes what needs to be documented in corresponding softifacts. The rational design process is not meant to be a linear process like the Waterfall model, but instead an iterative process using section stubs for information that is not yet available or not fully clear during the time of writing. Those stubs are then filled in over the development process, and existing documentation is updated so it appears to have been created in a linear fashion for ease of review later in the software lifecycle. The rational design process involves the following:

1. Establish/Document Requirements
2. Design and Document the Module Structure
3. Design and Document the Module Interfaces
4. Design and Document the Module Internal Structures
5. Write Programs
6. Maintain

Parnas provided a list of the most important documents [29] required for the rational design process, which Smith [**?**] [which paper was it? —DS] expanded upon by including complimentary artifacts such as source code, verification and validation plans, test cases, and build instructions. While there have been many proposed artifacts, the following curated list covers those most relevant to this thesis:

1. System Requirements

2. System Design Document
3. Module Guide
4. Module Interface Specification
5. Program Source Code
6. Verification and Validation Plan
7. Test cases
8. Verification and Validation Report
9. User Manual

This list is not exhaustive of all the types of softifacts, as there are many other design processes which use different types of softifacts. Looking at an agile approach using Scrum/Kanban, the softifacts tend to be distributed in different ways. Requirements are documented in tickets under so-called "epics", "stories", and "tasks" as opposed to a singular requirements artifact, and the acceptance criteria listed on those tickets make up the validation and verification plan.

Regardless of the process used, most attempt to document very similar information to that of the rational design process. Using the waterfall model as an example, we can see (Table 2.1) the rational design process and its artifacts map onto the model in a very straightforward manner.

As mentioned earlier, softifacts are important to development in a number of ways, such as easing the burden of maintenance and training. We outline the artifacts we are most interested below with a brief description of their purpose.

Table 2.1: Comparison of Rational Design Process and Waterfall Model

Rational Design phase	Corresponding Waterfall phase(s)	Common Softifacts
Establish/ Document Requirements	Requirements Analysis	System Requirements Specification Verification and Validation plan
Design and Document the Module Structure	System Design	Design Document
Design and Document the Module Interfaces	Program Design	Module Interface Specification
Design and Document the Module Internal Structures	Program Design	Module Guide
Write Programs	Coding	Source code Build instructions
Maintain	Unit & Integration Testing System Testing Acceptance Testing Operation & Maintenance	Test cases Verification and Validation Report

Name: **Software Requirements Specification**
Description: Contains the functional and nonfunctional requirements detailing what the desired software system should do.

Name: **System Design Document**
Description: Explains how the system should be broken down and documents implementation-level decisions that have been made for the design of the system.

Name: **Module Guide**
Description: In-depth explanation of the modules outlined in the System Design Document.

Name: **Module Interface Specification**
Description: Interface specification for each of the modules outlined in the System Design Document/Module Guide.

Name: **Program Source Code**
Description: The source code of the implemented software system.

Name: **Verification and Validation Plan**
Description: Uses the system requirements to document acceptance criteria for each requirement that can be validated.

Name: **Test cases**
Description: Implementation of the Verification and Validation Plan in source code (where applicable) or as a step-by-step guide for testers.

Name: **Verification and Validation Report**
Description: Report outlining the results after undertaking all of the testing initiatives outlined in the Verification and Validation plan and test cases.

Parnas [29] 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 2.2.

2.2 Software Reuse and Software Families

In this section we look at ways in which others have attempted to avoid "reinventing the wheel" by providing means to reuse software, in part or whole, and reuse the analysis and design efforts of those that came before.

2.2.1 Software/Program Families

Software/program families refer to a group of related systems sharing a common set of features, functionality, and design [34]. These systems are typically designed to serve a specific domain or market, and are often developed using a common set of core technologies and design principles.

One well-known example of a software family is the Microsoft Office suite. Each program in the suite is used for a specific application (word processing, spreadsheet management, presentation tools, etc), yet they all have similar design principles and user interface features. A user who understands how to use one of the software family members will have an intuitive sense of how to use the others thanks to the common

design features.

Another, much larger scale, software family is that of the GNU/Linux operating system (OS) and its various distributions. There are many variations on the OS depending on the user's needs. For an everyday computer desktop experience, there are general purpose distributions (Ubuntu, Linux Mint, Fedora, Red Hat, etc.). For server/data center applications, there are specialized server distributions (Ubuntu Server, Fedora Server, Debian, etc.). For systems running on embedded hardware there are lightweight, specialized, embedded distributions built for specific architecture (Armbian, RaspbianOS, RedSleeve, etc.). There are even specialized distributions that are meant to be run without persistent storage for specific applications like penetration/network testing (Kali Linux, Backbox, Parrot Security OS, BlackArch, etc.). However, if you are familiar with one particular flavour of linux, you'll likely be comfortable moving between several of the distributions built upon the same cores. You may even be familiar moving to other *NIX based systems like MacOS/Unix.

Software/program families can provide a range of benefits to both developers and end-users. For developers, software families can increase productivity by providing a reusable set of core technologies and design principles [34]. This can help reduce the time and effort required to develop new systems, and improve the quality and consistency of the resulting software. For end-users, program families provide a range of usability and functional benefits. Common features and UI elements improve the user experience by making it easier for users to learn and use multiple systems [3]. By also providing a range of related applications, such as those provided by the Microsoft office suite, software families help meet users' needs across a wider range of domains.

2.2.2 Software Reuse

Reusing software is an ideal means of reducing costs. If we can avoid spending time developing new software and instead use some existing application (or a part therein), we save time and money. There have been many proposals on ways to encourage software reuse, each with their own merits and drawbacks.

Component based software engineering (CBSE) is one such example. CBSE is an approach to software development that emphasizes using pre-built software components as the building blocks of larger systems. These reusable components can be developed independently and tested in isolation before being integrated into the larger system software.

One key benefit of CBSE, as mentioned above, is that it can help to reduce the cost and time required for software development, since developers do not need to implement everything from scratch. Additionally, CBSE can improve the quality and reliability of software systems, since components have typically been thoroughly tested and previously used in other contexts.

One CBSE framework is the Common Object Request Broker Architecture (CORBA), which provides a standardized mechanism for components to communicate with each other across a network. CORBA defines a set of interfaces and protocols that allow components written in different programming languages to interact with each other in a distributed environment [27].

Another CBSE framework is the JavaBeans Component Architecture. It is a standard for creating reusable software components in Java. JavaBeans are self-contained software modules with a well-defined interface that can be easily integrated into a variety of development environments and combined to form larger applications [28].

The largest challenge of CBSE is ensuring components are compatible with others and can be integrated into a larger system without conflicts or errors. In an effort to address this challenge, numerous approaches to component compatibility testing have been proposed [42].

Neighbors [24, 25, 26] proposed a method for engineering reusable software systems known as "Draco". [Finish talking about Draco here —DS]

2.2.3 Reproducible Research

Being able to reproduce results, is fundamental to the idea of good science. When it comes to software projects, there are often many undocumented assumptions or modifications (including hacks) involved in the finished product. This can make replication impossible without the help of the original author, and in some cases reveal errors in the original author's work [12].

Reproducible research has been used to mean embedding executable code in research papers to allow readers to reproduce the results described [36].

Combining research reports with relevant code, data, etc. is not necessarily easy, especially when dealing with the publication versions of an author's work. As such, the idea of *compendia* were introduced [10] to provide a means of encapsulating the full scope of the work. Compendia allow readers to see computational details, as well as re-run computations performed by the author. Gentleman and Lang proposed that compendia should be used for peer review and distribution of scientific work [10].

Currently, several tools have been developed for reproducible research including, but not limited to, Sweave [19], SASweave [21], Statweave [20], Scribble [8], and Org-mode [36]. The most popular of those being Sweave [36]. The aforementioned tools

maintain a focus on code and certain computational details. Sweave, specifically, allows for embedding code into a document which is run as the document is being typeset so that up to date results are always included. However, Sweave (along with many other tools), still maintains a focus on producing a single, linear document.

2.3 Literate Approaches to Software Development

There have been several approaches attempting to combine development of program code with documentation. Literate Programming and literate software are two such approaches that have influenced the direction of this thesis. Each of these approaches is outlined in the following sections.

2.3.1 Literate Programming

Literate Programming (LP) is a method for writing software introduced by Knuth that focuses on explaining to a human what we want a computer to do rather than simply writing a set of instructions for the computer on how to perform the task [17].

Developing literate programs involves breaking algorithms down into *chunks* [13] or *sections* [17] which are small and easily understandable. The chunks are ordered to follow a "psychological order" [33] if you will, that promotes understanding. They do not have to be written in the same order that a computer would read them. It should also be noted that in a literate program, the code and documentation are kept together in one source. To extract runnable code, a process known as *tangle* must be performed on the source. A similar process known as *weave* is used to extract and typeset the documentation.

There are many advantages to LP beyond understandability. As a program is developed and updated, the documentation surrounding the source code is more likely to be updated simultaneously. It has been experimentally found that using LP ends up with more consistent documentation and code [38]. There are many downsides to having inconsistent documentation while developing or maintaining code [18, 41], while the benefits of consistent documentation are numerous [11, 18]. Keeping the advantages and disadvantages of good documentation in mind we can see that more effective, maintainable code can be produced if properly using LP [33].

Regardless of the benefits of LP, it has not been very popular with developers [38]. However, there are several successful examples of LP's use in SC. Two such literate programs that come to mind are VNODE-LP [23] and "Physically Based Rendering: From Theory to Implementation" [32] a literate program and textbook on the subject matter. Shum and Cook [38] discuss the main issues behind LP's lack of popularity which can be summed up as dependency on a particular output language or text processor, and the lack of flexibility on what should be presented or suppressed in the output.

There are several other factors which contribute to LP's lack of popularity and slow adoption thus far. While LP allows a developer to write their code and its documentation simultaneously, that documentation is comprised of a single artifact which may not cover the same material as standard artifacts software engineers expect (see Section 2.1 for more details). LP also does not simplify the development process: documentation and code are written as usual, and there is the additional effort of re-ordering the chunks. The LP development process has some benefits such as allowing developers to follow a more natural flow in development by writing chunks in

whichever order they wish, keep the documentation and code updated simultaneously (in theory) because of their co-location, and automatically incorporate code chunks into the documentation to reduce some information duplication.

There have been many attempts to increase LP's popularity by focusing on changing the output language or removing the text processor dependency. Several new tools such as CWeb (for the C language), DOC++ (for C++), noweb (programming language independent), and others were developed. Other tools such as javadoc (for Java) and Doxygen (for multiple languages) were also influenced by LP, but differ in that they are merely document extraction tools. They do not contain the chunking features which allow for re-ordering algorithms.

With new tools came new features including, but not limited to, phantom abstracting [38], a "What You See Is What You Get" (WYSIWYG) editor [9], and even movement away from the "one source" idea [39].

While LP is still not mainstream [35], these more robust tools helped drive the understanding behind what exactly LP tools must do. In certain domains LP is becoming more standardized, for example: Agda, Haskell, and R support LP to some extent, even though it is not yet common practice. R has good tool support, with the most popular being Sweave [19], however it is designed to dynamically create up-to-date reports or manuals by running embedded code as opposed to being used as part of the software development process.

2.3.2 Literate Software

A combination of LP and Box Structure [22] was proposed as a new method called "Literate Software Development" (LSD) [2]. Box structure can be summarized as the

idea of different views which are abstractions that communicate the same information in different levels of detail, for different purposes. Box structures consist of black box, state machine, and clear box structures. The black box gives an external (user) view of the system and consists of stimuli and responses; the state machine makes the state data of the system visible (it defines the data stored between stimuli); and the clear box gives an internal (designer's) view describing how data are processed, typically referring to smaller black boxes [22]. These three structures can be nested as many times as necessary to describe a system.

LSD was developed with the intent to overcome the disadvantages of both LP and box structure. It was intended to overcome LP's inability to specify interfaces between modules, the inability to decompose boxes and implement the design created by box structures, as well as the lack of tools to support box structure [6].

The framework developed for LSD, "WebBox", expanded LP and box structures in a variety of ways. It included new chunk types, the ability to refine chunks, the ability to specify interfaces and communication between boxes, and the ability to decompose boxes at any level. However, literate software (and LSD) remains primarily code-focused with very little support for creating other software artifacts, in much the same way as LP.

2.4 Generative Programming

Generative programming is an approach to software development that focuses on automating the process of generating code from high-level specifications [5, 40]. By writing a program specification and feeding it to the generator, one does not have to manually implement the desired program.

One of the primary benefits of generative programming is that it can help to increase productivity and reduce the time and effort required to develop software [5]. By automating the generation of code, developers can focus on high-level design and specification, rather than low-level implementation details.

Generative programming has the added benefit of helping to improve the quality of software by reducing the risk of errors and inconsistencies [5, 40]. Since the code is generated automatically from high-level specifications, there is less room for human error, and the generated code is typically more consistent and predictable.

There are also some potential drawbacks to generative programming. For instance, the generated code may not always be optimal or efficient [5, 40].As the code is generated automatically, it may not take into account all of the nuance or complexity of the underlying system potentially leading to suboptimal performance or other issues.

Generative programming also requires a significant upfront investment in time and effort to develop the generators and other tools needed to automate the process of code generation [5, 40]. This means it is often not worth the effort to use generative programming for one-off projects.

There are a large number of generative programming tools available today. Some, like template metaprogramming (TMP) tools, are built into a number of programming languages (C++, Rust, Scala, Java, Go, Python, etc.) and offer varying levels of support for generative programming. [Give some examples of code generators and their applications here —DS]

Table 2.2: 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	Software Architects QA analysts	Define exactly what specification the software system must adhere to.
Module Guide	All developers QA analysts	Outline implementation decisions around the separation of functionality into modules and give an overview of each module.
Module Interface Specifications	Developers who implement the module Developers who use the module QA analysts	Detail the exact interfaces of the modules from the Module Guide.
Source Code / Executable	All developers QA analysts End users	Implements the machine instructions designed to address the overall problem for which the software system has been specified
Verification and Validation Plan	Developers who implement the module QA analysts	Describe how the software should be verified using tests that can be validated. Includes module-specific and system-wide plans.

Chapter 3

A look under the hood:

Our process

[Make sure we talk about continuous integration / git processes / etc. Actually might belong in the next chapter - iteration and refinement —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.

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 Section 1.2, 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. The following cards 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 - Fill in once all examples in the thesis are done —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 remodeling 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. Relevant artifacts: [TODO —DS]

Name: **Projectile**
Problem being solved: A system is needed to efficiently and correctly predict the landing position of a projectile. 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. 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.[1] 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.

The following subsections present a brief sampling of our process of breaking down softifacts, acknowledging that a comprehensive overview would be excessively lengthy.

[1]Refer to Section 2.1 for a general summary of softifacts.

3.2.1 SRS

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 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 GlassBR. 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 [will we actually? depends on length of chapter —DS], for now we will try to ignore any superficial differences (spelling, grammar, phrasing, etc.) in each of them while we look for commonality. We are also trying to determine how the non-superficial differences relate to the document template, general problem domain, and specific system information.

Looking at the (truncated for space) Table of Symbols, Table of Units, and Table of Abbreviations and Acronyms sections (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)

1. Reference Material
   1.1. Table of Units
   1.2. Table of Symbols
   1.3. Abbreviations and Acronyms
2. Introduction
   2.1. Purpose of Document
   2.2. Scope of Requirements
   2.3. Characteristics of Intended Reader
   2.4. Organization of Document
3. Stakeholders
   3.1. The Customer
   3.2. The Client
4. General System Description
   4.1. System Context
   4.2. User Characteristics
   4.3. System Constraints
5. Specific System Description
   5.1. Problem Description
      5.1.1. Physical System Description
      5.1.2. Goal Statements
   5.2. Solution Characteristics Specification
      5.2.1. Assumptions
      5.2.2. Theoretical Models
      5.2.3. General Definitions
      5.2.4. Data Definitions
      5.2.5. Instance Models
      5.2.6. Data Constraints
      5.2.7. Properties of a Correct Solution
6. Requirements
   6.1. Functional Requirements
   6.2. Non-Functional Requirements
7. Likely Changes
8. Unlikely Changes
9. Traceability Matrices and Graphs
10. Values of Auxiliary Constants
11. References
12. Appendix

Figure 3.1: The Table of Contents from the ([expanded? —DS]) Smith et al. template

1.1 Table of Units

The unit system used throughout is SI (Système International d'Unités). In addition to the basic units, several derived units are also used. For each unit, the Table of Units lists the symbol, a description and the SI name.

Symbol	Description	SI Name
kg	mass	kilogram
m	length	metre
N	force	newton
Pa	pressure	pascal
s	time	second

(a) Table of Units Section

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? Refer to Section 2.1, for more details. A brief summary is available in Table 2.2.

Returning to our exercise of breaking down each section of the SRS to determine the subtleties of *what* is contained therein[2] it should be unsurprising that each section maps to the definition provided in the Smith et al. template. However, as noted

[2]The breakdown details are omitted for brevity and due to their monotonous nature, although the overall process is very much akin to the breakdown of the Reference Material section.

1.2 Table of Symbols

The symbols used in this document are summarized in the Table of Symbols along with their units. The symbols are listed in alphabetical order.

Symbol	Description	Units
a	Plate length (long dimension)	m
AR	Aspect ratio	–
AR_{max}	Maximum aspect ratio	–
B	Risk of failure	–
b	Plate width (short dimension)	m
capacity	Capacity or load resistance	Pa
d_{max}	Maximum value for one of the dimensions of the glass plate	m
d_{min}	Minimum value for one of the dimensions of the glass plate	m
E	Modulus of elasticity of glass	Pa

(b) Table of Symbols (truncated) Section

1.3 Abbreviations and Acronyms

Abbreviation	Full Form
A	Assumption
AN	Annealed
AR	Aspect Ratio
DD	Data Definition
FT	Fully Tempered
GS	Goal Statement
GTF	Glass Type Factor
HS	Heat Strengthened
IG	Insulating Glass
IM	Instance Model
LC	Likely Change
LDF	Load Duration Factor
LG	Laminated Glass

(c) Table of Abbreviations and Acronyms (truncated) Section

Figure 3.2: The reference sections of GlassBR

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-system-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.

Observing the contents of an SRS template adhere to said template may seem mundane, but it is a necessary step before we can move on to other softifacts. Without understanding what the SRS template intends to convey it is hard to assess weather or not the case study SRS conveys that information. With that in mind, we can move on to the MG and source code.

[Current plan for following subsections: Brief description of the softifact, show an example of similarities within (ex. MG/MIS have a section per module, each section is organized the same way, some are filled in, some aren't), then follow a requirement through the MG to something in the MIS and finally to code. We'll dissect differences between case studies when looking at the patterns in Section 3.3. This also plants the seeds of "see, there's the same info moving from SRS → MG → MIS** → Code without stating it explicitly, which we can then do in the pattern section. —DS]

[Example to use should be a DD/IM from GlassBR, goes to calculations module in the MG, and finally a method in the source code —DS]

3.2.2 Module Guide

The module guide (MG) is a softifact that details the architecture of a given software system. It holds a number of design decisions around sensibly grouping functionalities within the system into modules to fulfill the requirements laid out in the SRS. For

example, one might have an input/output module for handling user input and giving the user feedback through the display (ie. via print commands or some other output), or a calculations module that contains all of the calculation functions being performed in the normal operation of the given software system. The Smith et al. MG template also includes a traceability matrix for ease of verifying which requirements are fulfilled by which modules. Finally, the MG includes considerations for anticipated or unlikely changes that the system may undergo during its lifecycle.

Figure 3.3 shows the table of contents for the GlassBR case study's MG. For the sake of brevity we will omit the other case studies here (they can be found at [TODO —DS]). Just as with the SRS we are looking for commonality and understanding of what the document is trying to portray to the reader. As such we will ignore superficial differences between the MG sections. As the MG is a fairly short document we will look at each of the most relevant sections as part of this exercise.

Breaking down the MG by section, we can see that the introduction is itself completely generic boilerplate explaining the purpose of the MG, the audience, and some references to other works that explain why we would make certain choices over other (reasonable) ones given the opportunity. There is nothing system-specific, nor specific to the given problem domain of the case study.

Following through the table of contents into the "Anticipated and Unlikely Changes" section, we see that again the introductions to this section and its subsections are generic boilerplate, however the details of each section are not. Both subsections are written in the same way: as a list of labeled changes (AC# for anticipated change, UC# for unlikely change). This is the first place we see both problem-domain and system-specific information Interestingly, the Module Hierarchy section follows the

Module Guide for GlassBR

Spencer Smith and Thulasi Jegatheesan

July 25, 2018

Contents

Figure 3.3: Table of Contents for GlassBR Module Guide

5.2.4 Calc Module (M5)

Secrets: The equations for predicting the probability of glass breakage, capacity, and demand, using the input parameters.

Services: Defines the equations for solving for the probability of glass breakage, demand, and capacity using the parameters in the input parameters module.

Implemented By: GlassBR

Figure 3.4: Calc Module from the GlassBR Module Guide

same general style: it is a list of modules which represent the leaves of the module hierarchy tree and each one is labeled (M#).

Skipping ahead to the module decomposition, we find a section heading for each Level 1 module in the hierarchy, followed by subsections describing the Level 2 modules. The former are almost entirely generic boilerplate (for example common Level 1 modules include: Hardware-Hiding, Behaviour-Hiding, and Software Decision modules), but the latter are problem-domain or system specific. An example of a system-specific module is shown in Figure 3.4.

Each module is described by its secrets, services, and what it will be implemented by. For example, a given module could be implemented by the operating system (OS), the system being described (ex. GlassBR), or a third party system/library that will inter-operate with the given system.

Finally we have a traceability matrix and use hierarchy diagram. Both are visual representations of how the different modules implement the requirements and use each other respectively. The traceability matrix provides a direct and obvious link between the SRS and MG, where other connections between the two softifacts have been implicit until this point. Generally, the next softifact would be the MIS, however

as it is structured so similarly to the MG (one section per module, each section organized in a very similar way, a repeated use hierarchy, etc) we will skip it for brevity. The MIS includes novel system-specific, implementation-level information denoting the interfaces between modules, but for our current exercise does not provide any revelations beyond that of the MG.

The MG gives us a very clear picture of *decisions* made by the system designers, as opposed to the knowledge of the system domain, problem being solved, and requirements of an acceptable solution provided in the SRS. The MG provides platform and implementation-specific decisions, which will eventually be translated into implementation details in the source code. With that in mind, let us move on to the source code.

3.2.3 Source Code

The source code is arguably the most important softifact in any given software system since it serves as the set of instructions that a computer executes in order to solve the given problem. With only the other softifacts and without the source code, we would have a very well defined problem and acceptance criteria for a possible solution, but would never actually solve the problem.

As the source code is the executable set of instructions, one would expect it to be almost entirely system and problem-domain specific with very little boilerplate. Looking into the source of our case studies, we find this to be mostly true barring the most generic of library use (ex. `stdio` in C).

Returning to our example of the MG from GlassBR (Figure 3.3) and comparing it to the python source code structure shown in Figure 3.5 we can see that the

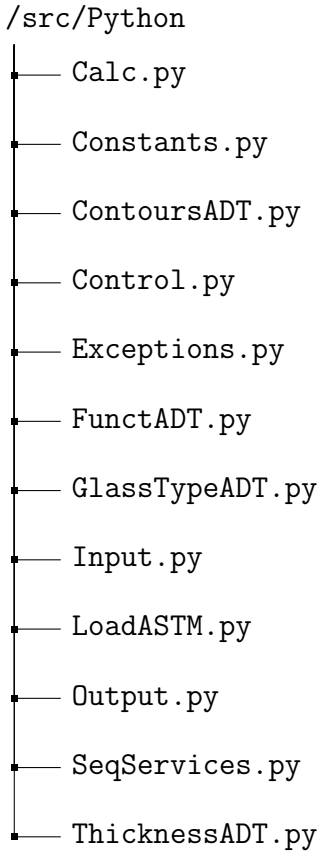

Figure 3.5: Python source code directory structure for GlassBR

source code follows almost identically in structure to the module decomposition. The only difference being the existence of an exceptions module defining the different types of exceptions that may be thrown by the other modules. While this can be considered a fairly trivial difference, likely made for ease of maintenance, readability, and extensibility, it highlights that the two softifacts are out of sync. We speculate this difference was caused by a change made during the implementation phase, wherein the MG was not updated to reflect the addition of an exceptions module.

Let us look deeper into the code for one specific module, for example the Calc module introduced in the MG (Figure 3.4). The source code for said module can be found in Figure 3.6. In the source code we see a number of calculation functions, including those that calculate the probability of glass breakage, demand (also known as *load* or q), and capacity (also known as *load resistance* or LR) as outlined in the *secrets* section of the Calc module definition in the MG. We also see a number of intermediary calculation functions required to calculate these values (for example `calc_NFL` and its dependencies).

The source code provides clear instructions to the machine on how to calculate each of these values and their intermediaries; it provides the actionable steps to solve the given problem. When we compare the code with relevant sections of the SRS, specifically the Data Definitions (DDs) for each term, we can see a very obvious transformation from one form to the other; the symbol used by the DD is the (partial) name of the function in the source code and the equation from the DD is calculated within the source code. This is one of many patterns we see across our softifacts within each case study.

```
5  from Input import *
6  from ContoursADT import *
7  from math import log, exp
8
9  ## @brief Calculates the Dimensionless load
10 #  @return the unitless load
11 def calc_q_hat( q, params ):
12     upper = q * (params.a * params.b) ** 2
13     lower = params.E * (params.h ** 4) * params.GTF
14     return ( upper / lower )
15 ## @brief Calculates the Stress distribution factor based on Pbtol
16 #  @return the unitless stress distribution factor
17 def calc_J_tol( params ):
18     upper1 = 1
19     lower1 = 1 - params.Pbtol
20
21     upper2 = (params.a * params.b) ** (params.m - 1)
22     lower2 = params.k * ((params.E * (params.h ** 2)) ** (params.m)) * params.LDF
23
24     return (log( (log(upper1 / lower1)) * (upper2 / lower2) ))
25 ## @brief Calculates the Probability of glass breakage
26 #  @return unitless probability of breakage
27 def calc_Pb( B ):
28     output = 1 - (exp(-B))
29     if not (0 < output < 1):
30         raise InvalidOutput("Invalid output!")
31     return (output)
32 ## @brief Calculates the Risk of failure
33 #  @return unitless risk of failure
34 def calc_B( J, params ):
35     upper = params.k * ((params.E * (params.h) ** 2) ** params.m) * params.LDF * exp(
       J)
36     lower = ((params.a * params.b) ** (params.m - 1))
37     return ( upper / lower )
38 ## @brief Calculates the Non-factores load
39 #  @return unitless non-factored load
40 def calc_NFL( q tol , params ):
41     upper = q tol * params.E * (params.h ** 4)
42     lower = (params.a * params.b) ** 2
43     return ( upper / lower )
44 ## @brief Calculates the Load resistance
45 #  @return unitless load resistance
46 def calc_LR( NFL, params ):
47     return ( NFL * params.GTF * params.LSF )
48 ## @brief Calculates Safetey constraint 1
49 #  @return true if the calculated probability is less than the tolerable probability
50 def calc_is_safePb( Pb, params ):
51     if (Pb < params.Pbtol):
52         return True
53     else:
54         return False
55 ## @brief Calculates Safetey constraint 2
56 #  @return true if the load resistance is greater than the load
57 def calc_is_safeLR( LR, q ):
58     if (LR > q):
59         return True
60     else:
61         return False
```

Figure 3.6: Source code of the Calc.py module for GlassBR

3.3 Identifying Repetitive Redundancy

From the examples in Section 3.2, we can see a number of simple patterns emerging with respect to organization and information repetition within a case study. Upon applying our process to all of the case studies and adopting a broader perspective, numerous instances emerge where patterns transcend individual case studies and remain universally applicable. Several of these patterns should be unsurprising, as they relate to the template of a particular softifact. It is interesting, however, that patterns of information organization crop up within a given softifact in multiple places, containing distinct information.

Returning to our example from Section 3.2.1, 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 defining terms with respect to their symbolic representation and general information relevant to those terms. 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 observe that the introduction to the Table of Units is nothing more than boilerplate text dropped into each case study verbatim; it is completely generic and applicable to *any* software system using SI units. The introduction to the Table of Symbols also appears to be boilerplate across several examples, however, it does have minor variations which we can see by comparing Figure 3.2b to Figure 3.7 (GlassBR compared to GamePhysics). These variations reveal the obvious: the variability between systems is greater than simply a difference in choice of symbols, and so there is some

1.2 Table of Symbols

The table that follows summarizes the symbols used in this document along with their units. More specific instances of these symbols will be described in their respective sections. Throughout the document, symbols in **bold** will represent vectors, and scalars otherwise. The symbols are listed in alphabetical order.

symbol	unit	description
a	$\mathrm{m\,s^{-2}}$	Acceleration
α	$\mathrm{rad\,s^{-2}}$	Angular acceleration
C_{R}	unitless	Coefficient of restitution
F	N	Force
g	$\mathrm{m\,s^{-2}}$	Gravitational acceleration ($9.81\ \mathrm{m\,s^{-2}}$)
G	$\mathrm{m^3\,kg^{-1}\,s^{-2}}$	Gravitational constant ($6.673 \times 10^{-11}\ \mathrm{m^3\,kg^{-1}\,s^{-2}}$)
I	$\mathrm{kg\,m^2}$	Moment of inertia

Figure 3.7: Table of Symbols (truncated) Section from GamePhysics

system-specific knowledge being encoded. While we can intuitively infer this conclusion based solely on each system addressing a different problem, our observation of the (structural) patterns within this SRS section confirms it.

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 one or more 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

Number	DD7
Label	**Dimensionless Load ($\hat{q}$)**
Equation	$\hat{q} = \frac{q(ab)^2}{Eh^4 \text{GTF}}$
Description	q is the 3 second equivalent pressure, as given in IM3 a, b are dimensions of the plate, where $(a > b)$ E is the modulus of elasticity h is the true thickness, which is based on the nominal thickness as shown in DD2 GTF is the Glass Type Factor, as given by DD6
Source	[2], [8, Eq. 7]
Ref. By	DD4

Figure 3.8: Data Definition for Dimensionless Load ($\hat{q}$) from GlassBR SRS

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 that look identical, but mean different things, for example: PM can refer to a Product Manager, Project Manager, Program Manager, Portfolio Manager, etc.

By continuing to breakdown the SRS and other softifacts, we are able to find many more patterns of knowledge repetition. For example, we see the same concept being introduced in multiple areas within a single artifact and across artifacts in a project. Figure 3.8 shows the data definition for $\hat{q}$ in GlassBR. That same term was previously defined with fewer details in the table of symbols (omitted here for brevity), as well as showing up implicitly or in passing in the MG (Figure 3.4 and the *loadASTM* module respectively), and implemented in the Source Code (Figure 3.6 lines 11-14). It should be noted that the SRS contains many references to $\hat{q}$, such as in the data definitions of the Stress Distribution Factor (J) and Non-Factored Load (NFL). There are also

implied references through intermediate calculations, for example the Calculation of Capacity (LR) is defined in terms of NFL which relies on $\hat{q}$.

Although the full definition of $\hat{q}$ is initially provided for a human audience only once, it is necessary to reference it in different ways for different audiences. Each audience is expected to grasp the symbol's meaning within their given context or consult other softifacts for more comprehensive understanding. When reading the SRS, the data definitions and other reference materials play a crucial role in swiftly comprehending the complete definition of $\hat{q}$ in relation to the system's inputs, outputs, functional requirements, and acceptance criteria.

The MG, on the other hand, briefly mentions $\hat{q}$ when defining the responsibilities of both the *loadASTM* and *Calc* modules (the former being responsible for loading values from a file, and the latter utilizing those values for calculations), whereas the source code provides a highly detailed definition to ensure accurate execution of the relevant calculation(s).

The varying level of detail across the softifacts should not come as a surprise since each softifact targets a different audience and their specific needs at various stages of the software development process. Although the level of verbosity may differ, the core information remains consistent: the authors are consistently referring to the definition of $\hat{q}$ via its symbolic representation, regardless of the level of detail incorporated. The goal is to convey relevant aspects of knowledge of a given term, while eliding that which is deemed superfluous, based on the context and the specific requirements of our audience. In other words, the authors only *project* some portion of their knowledge of given terms at a given time, depending on their needs (precision,

brevity, clarity, etc.), the expectations of the audience, and contextual relevance. [3] The audience, on the other hand, engages in *knowledge transformation*, whereby they consume the representation (projected knowledge) and transform it into their own internal representation, based on their personal knowledge-base.

Relying on common representations, eliding definitions, projecting and transforming knowledge are fundamental to the way humans communicate. They are readily observable in all forms of communications, whether written or oral, as we assign meaning to given sounds and symbols (words) according to the agreed upon grammar of a given language and use those words (knowledge projections) to simplify communication to a given audience. A context-specific glossary, or more generally a dictionary, is a prime example of a knowledge-base that we use for communication via knowledge projections and transformations. By maintaining a shared vocabulary, we can communicate using the symbolic representations (words) instead of requiring terms to be decomposed (defined) to their most basic form. However, communication of this sort is still imperfect, due to gaps in shared knowledge between participants or misunderstanding of overloaded terms. Interpersonal communications can involve nuance and context-dependent interpretations, yet they still boil down to knowledge projection on the part of the communicator and knowledge transformation on the part of the communicatee. The latter can infer context, or be provided with explicit context, which affirms their use of the appropriate knowledge transformations.

Returning to the context of software systems, if we broaden our view from a single system, to a software family, we can also find patterns of commonality and repeated knowledge across the various softifacts of the family members (For example

[3] We have only referred to the term as $\hat{q}$ in this section to emphasize our argument and make a meta-argument that the definition is irrelevant to our audience in this example. What matters is the symbolic reference, which we share a common understanding of.

Number	T1
Label	**Conservation of thermal energy**
Equation	$-\nabla \cdot \mathbf{q} + g = \rho C \frac{\partial T}{\partial t}$
Description	The above equation gives the conservation of energy for transient heat transfer in a material of specific heat capacity C ($\mathrm{J\,kg^{-1}\,°C^{-1}}$) and density ρ ($\mathrm{kg\,m^{-3}}$), where $\mathbf{q}$ is the thermal flux vector ($\mathrm{W\,m^{-2}}$), g is the volumetric heat generation ($\mathrm{W\,m^{-3}}$), T is the temperature (°C), t is time (s), and ∇ is the gradient operator. For this equation to apply, other forms of energy, such as mechanical energy, are assumed to be negligible in the system (A1). In general, the material properties (ρ and C) depend on temperature.
Source	http://www.efunda.com/formulae/heat_transfer/conduction/overview_cond.cfm
Ref. By	GD2

Figure 3.9: Theoretical Model of conservation of thermal energy found in both the SWHS and NoPCM SRS

the SWHS and NoPCM case studies) as they have been developed to solve similar, or in our case nearly identical, problems. Software family members are good examples to help determine what types of information or knowledge provided in the softifacts belong to the system-domain, problem-domain, or are simply general (boilerplate).

Looking at SWHS and NoPCM, we can easily find identical theoretical models (TMs) as the underlying theory for each system is based on the problem domain (see example in Figure 3.9). However, when we follow the derivations from the TMs to the Instance Models (IMs), we find the resulting equations have changed due to the context of the system; the lack of PCM has changed the relevant equations for calculating the energy balance on water in the tank as shown in Figure 3.10.

While the above examples are fairly small and specific, they are indicative of a larger, more generalizable, set of patterns of knowledge organization and repetition. These patterns are at their core: the use of common knowledge that has been

Number	IM1
Label	**Energy balance on water to find T_W**
Input	m_W, C_W, h_C, A_C, h_P, A_P, t_{final}, T_C, T_{init}, $T_P(t)$ from IM2 The input is constrained so that $T_{\text{init}} \leq T_C$ (A11)
Output	$T_W(t)$, $0 \leq t \leq t_{\text{final}}$, such that $\frac{dT_W}{dt} = \frac{1}{\tau_W}[(T_C - T_W(t)) + \eta(T_P(t) - T_W(t))]$, $T_W(0) = T_P(0) = T_{\text{init}}$ (A12) and $T_P(t)$ from IM2
Description	T_W is the water temperature (°C). T_P is the PCM temperature (°C). T_C is the coil temperature (°C). $\tau_W = \frac{m_W C_W}{h_C A_C}$ is a constant (s). $\eta = \frac{h_P A_P}{h_C A_C}$ is a constant (dimensionless). The above equation applies as long as the water is in liquid form, $0 < T_W < 100$°C, where 0°C and 100°C are the melting and boiling points of water, respectively (A14, A19).
Sources	[4]
Ref. By	IM2

(a) SWHS Instance Model for Energy Balance on Water

Number	IM1
Label	**Energy balance on water to find T_W**
Input	m_W, C_W, h_C, A_C, t_{final}, T_C, T_{init} The input is constrained so that $T_{\text{init}} \leq T_C$ (A9)
Output	$T_W(t)$, $0 \leq t \leq t_{\text{final}}$, such that $\frac{dT_W}{dt} = \frac{1}{\tau_W}(T_C - T_W(t))$, $T_W(0) = T_{\text{init}}$
Description	T_W is the water temperature (°C). T_C is the coil temperature (°C). $\tau_W = \frac{m_W C_W}{h_C A_C}$ is a constant (s). The above equation applies as long as the water is in liquid form, $0 < T_W < 100$°C, where 0°C and 100°C are the melting and boiling points of water, respectively (A10).
Sources	Original SRS with PCM removed
Ref. By	

(b) NoPCM Instance Model for Energy Balance on Water

Figure 3.10: Instance Model difference between SWHS and NoPCM

projected through some means, and patterns of organization of those knowledge projections within softifacts. Common knowledge, in this case, refers to one of three categories of knowledge: system-specific, domain-specific, or common to softifacts as a whole. It should also be noted that knowledge projections may include the identity projection (ie. the full, unabridged definition) as they are dependent on the relevance to the audience of the given softifact. Regardless, the captured knowledge fundamentally underlies these patterns of repetition, and is where we need to focus if we intend to reduce unnecessary redundancy.

3.4 Organizing knowledge - a fluid approach

Given the knowledge categories and patterns of use across softifacts we have seen in the previous section, we can generalize knowledge projections by their projection functions. [What follows is very rough and may need some definitions/tweaking to be more clear, but it makes sense in my head —DS] Identity projection functions directly repeat knowledge verbatim i.e. $p(K) \equiv K$ for some piece of knowledge k [I think I'll remove all of the set notation here, it comes out of nowhere and probably won't come up anywhere else or be defined anywhere so it's a bit jarring. I'm leaving it in for now because it makes sense to me —DS]. For example a copy-and-paste approach would be an identity projection. It should be noted in our case that as long as the projection used contains the full definition and context of a piece of knowledge, that projection is considered an identity projection regardless of changes to notation (ex. "$x = y$" vs "$y = x$") or language (ex. "$x = y$" vs "x is equal to y" vs "x est égal à y"). [Please correct my French if I'm wrong here —DS] Non-identity projections require using representations that elide some details of the knowledge at hand to make it

more palatable for the audience of the given softifact i.e. $p(K) \subset K$. Similarly, we can have multivariant projection functions (whether identity or not) which project knowledge from several places into one form, i.e. $p(K, L, M) \subseteq K \cup L \cup M$.

In both cases, we have a knowledge core that is fully defined and then we apply a projection function to retrieve the necessary information for our softifact. We can postulate that organizing our knowledge cores into some type of structure, with some assortment of projection functions (ex. Looking up the full definition would be done through an identity projection function) will allow us to reduce the need for manual duplication and remove unnecessary redundancies, as anything we need to include in our softifacts can be retrieved from a given source with a given projection function.

Keeping in mind that our core knowledge is used across all softifacts via projections, we may naively choose to consolidate all knowledge cores into one database. This naive approach works well enough for a limited set of examples, but it quickly becomes apparent (refer to the PM example from Section 3.3) that context is highly important and the sheer scope of knowledge to be organized may become unwieldy. After breaking down multiple case studies, we believe collecting knowledge cores into categories based on their domain(s) is a more easily navigable and maintainable approach. This also allows us to keep some context information at a meta-level (ex. Physics knowledge would be categorized into a Physics knowledge-base). Then for any given system, we would likely only need to reference across a handful of contexts (knowledge-bases) relevant to the domain.

There is knowledge fundamental to all softifacts as it is contextual to the domain of softifact writing itself. This kind of meta-knowledge would be useful to have readily available in its own knowledge-base. The same could be said for things like SI

Unit definitions, while they only apply to measuring physical properties, we see some domains built off of physics that operate at a higher level of assumed understanding (ex. Chemistry abstracts some of the physics details, while being directly reliant on them).

Some of the knowledge used in our softifacts is derived from other, more fundamental, knowledge cores. For example, when using SI units, we may choose to use a derived unit (newton, joule, radian, etc.) which is a better fit for the application domain of the system being documented. While we want to avoid unnecessary redundancy, we can argue that derived units are good candidates for acceptable redundancy. For example, if anywhere we use *Joules* we replace that with the definition ($J = \frac{kg \cdot m^2}{s^2}$) we then run into a problem of context and complexity. Generally, the audience for a given softifact will have an internal representation of context-specific knowledge, so even something as straightforward as changing the units from J to $\frac{kg \cdot m^2}{s^2}$ will put unnecessary load on said audience and force them to engage in more intensive knowledge transformations, while also potentially making the softifacts harder to parse for experts in the domain. In these cases, we want to use the derived knowledge in place of the core knowledge.

Being able to specify our level of abstraction through the progressive application of projection functions eludes to another necessary piece of knowledge organization: the projection functions themselves. As we project out core knowledge that we know of as otherwise commonly derived concepts (like the Joule example above), we should also like to store them. For example, derived units may end up in the same context as the SI Units, defined by specific projection functions applied to SI Units. Continuing the Joule example, we would be applying a projection function across core knowledge

related to energy and specific SI Units, then calling that projection *Joule* and giving it a symbolic representation J that we can refer to later.

We want to take a fluid and practical approach to organizing knowledge, such that we can keep domain-related knowledge cores together with useful derivations. We want to separate knowledge in unrelated domains, such that it is straightforward to look up whatever we need with relative ease. The specific implementation for organization will be detailed later (See Chapter 4.2).

[Next section needs to summarize "we need to capture knowledge", "store knowledge", "project knowledge", and have the framework to support that across multiple softifact domains and multiple code langs —DS]

3.5 Summary - The seeds of Drasil

Through this chapter, as part of our effort to reduce unnecessary redundancy across the software development process, we have taken an approach to breaking down softifacts to the core knowledge they present and looked for commonalities in that knowledge between them. We use several case study systems that fit our scope (input $\rightarrow$ process $\rightarrow$ output) as examples to give a concrete, applied base to the work.

Generally, we see softifacts for a given system have a lot in common, namely they require the same core knowledge tailored to a specific audience for each softifact. This knowledge is organized in a meaningful way, and portions relevant to the context of the softifact are presented to the audience.

We delved into the idea of knowledge cores and projection functions for producing context-relevant pieces of knowledge that are consumable by a given audience. We have also explored strategies for organizing that knowledge in a practical manner.

We have determined the three main components necessary for any useful softifact: knowledge, context (ie. audience), and organizational structure. From here we can operationalize each component in a reusable and (relatively) redundancy-free manner. This operationalization informs the initial design for our framework Drasil which will be covered in depth in Chapter 4.

Effectively, we want to automate the generation of softifacts through applying projections to knowledge and presenting it in a given structure. The structure of softifacts is relatively straightforward to deal with, we can use templates, blueprints, or deterministic generation which rely on relatively common technologies. The knowledge-capture and projection is much more interesting as it relies on some yet-to-be-determined knowledge-capture mechanism that can provide us with chunks (borrowing the term from LP) of knowledge that can then be fed to projection functions in some context-aware manner.

[Want to work in something about "our framework needs to be developed with consistency in mind, so we take a practical, example-driven (case studies) approach to minimize introducing new errors and inconsistencies." Could also fit in Drasil section, but I feel like introducing it here would be better. —DS]

Chapter 4

Drasil

[**Section Roadmap: – This is where the real meat of Drasil is discussed (implementation details) – Intro to our knowledge-capture mechanisms - Chunks/hierarchy - Break down each with examples from the case studies. - Look for 'interesting' examples (synonyms, acronyms, complexity, etc.) – Intro to the DSL - Captured knowledge is useless without the transformations/rendering engine - DSL for each softifact —DS]

In this chapter we 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. The name Drasil, derived from Norse Mythology's world tree Yggdrasil whose branches spread across the many realms, is representative of how our framework spreads across the many domains and contexts relevant to software generation.

4.1 Drasil in a nutshell

Manually writing and maintaining a full range of softifacts is redundant, tedious, and often leads to divergent softifacts. Drasil is a purpose-built framework created 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 softifacts.

However, Drasil is not, nor is it intended to be, a panacea for all the woes of software development. Even the seemingly well-defined problems of unnecessary redundancy and manual duplication turn out to be large, many-headed beasts which exist 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 acceptable margins for error.

With Drasil being built around this specific class of problems, we remain aware that there are likely many in-built assumptions in its current state that could affect its applicability to other domains. As such, we consider expanding Drasil's reach as 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. We must make it clear 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

For Drasil to function as intended, we need a means of capturing and organizing the underlying knowledge of the software systems we are trying to build alongside common knowledge that would be relevant across our entire target domain of SC software. This

```
33 -- === DATA TYPES/INSTANCES === --
34 -- | Used for anything worth naming. Note that a 'NamedChunk' does not have an
       acronym/abbreviation
35 -- as that's a 'CommonIdea', which has its own representation. Contains
36 -- a 'UID' and a term that we can capitalize or pluralize ('NP').
37 --
38 -- Ex. Anything worth naming must start out somewhere. Before we can assign equations
39 -- and values and symbols to something like the arm of a pendulum, we must first give
       it a name.
40 data NamedChunk = NC {
41   _uu :: UID,
42   _np :: NP
43 }
```

Figure 4.1: NamedChunk Definition

knowledge capture method must also be robust enough to be operationalized by a multi-faceted generation framework.

Before we can design a knowledge capture mechanism, we must first define what exactly we believe we need to capture. It is nearly impossible to consider every case of knowledge that could be used in the domain of SC software, not to mention an extremely large undertaking to begin with "all the things". As such, we have decided to take an iterative, progressive refinement approach to our knowledge capture mechanisms which should be no surprise and follows from our general process for developing the Drasil framework.

4.2.1 Capturing Knowledge via Chunks

We begin by borrowing and re-purposing the *chunk* term from Literate Programming (LP) and use it to create a simplified, extensible, and ever expanding hierarchy of chunks based on the requirements for capturing a single piece of knowledge. Our base chunk, `NamedChunk` is defined in Figure 4.1 and can be thought of as any uniquely identifiable term. It is composed of a `UID`, or unique identifier, and a `NP`, or noun phrase, representing the term.

```
probability  = dcc "probability"  (cnIES "probability")          "The likelihood of
    an event to occur"
rate         = dcc "rate"         (cn' "rate")                   "Ratio that compares
     two quantities having different units of measure"
rightHand    = dcc "rightHand"    (cn' "right-handed coordinate system")  "A
    coordinate system where the positive z-axis comes out of the screen."
shape        = dcc "shape"        (cn' "shape")                  "The outline of an
    area or figure"
surface      = dcc "surface"      (cn' "surface")                "The outer or
    topmost boundary of an object"
unit_        = dcc "unit"         (cn' "unit")                   "Identity element"
vector       = dcc "vector"       (cn' "vector")                 "Object with
    magnitude and direction"
```

Figure 4.2: Some example instances of `ConceptChunk` using the `dcc` smart constructor

A single node does not a hierarchy make, but now with the root `NamedChunk` defined, we can begin to progressively extend it to cover any new chunk types we may need for our knowledge capture requirements. For example, if we want to capture a simple term with its definition, we need more than just a `NamedChunk`. We need to extend `NamedChunk` to include a definition, and we refer to this particular variant chunk as a `ConceptChunk`. For ease of creation, we define a number of semantic, so-called *smart constructors* for each chunk type which allow us to more simply define our chunks and their intermediary datatypes (ie: `UID` and `NP` from the `NamedChunk` example). An example of such smart constructors being used to create some simple `ConceptChunk` instances can be seen in Figure 4.2. Note there are smart constructors for each datatype when multiple variants exist and could be used in a given place. Looking at the `ConceptChunk` example, we can see two different smart constructors (`cnIES` and `cn'`) being used to create the `NP` instance. Both smart constructors create an instance of `NP`, but in this case the smart constructor used defines given properties (ie. pluralization rules of the noun phrase used as a term) for the instance to simplify construction.

These simple chunks are fine as a starting point, however, knowing the SC domain we can already foresee the need for a variety of other chunks. For brevity, we will only expand upon the chunks needed for the examples used in this paper. More information, including detailed definitions of all the types and current state of Drasil can be found in the wiki (https://github.com/JacquesCarette/Drasil/wiki/) or the Haddock documentation which can be generated from the source code or found online at https://jacquescarette.github.io/Drasil/docs/full/

4.2.2 Chunk Combinatorics

[Would we call out chunk hierarchy a DSL? Or would the DSL be more along the lines of Expr and Sentence which make up our chunks? —DS]

Even with extremely simple chunks we have seen the need for multiple chunk types and a number of ways to construct instances of those types. The more we extend our chunk hierarchy, the larger the number of possible combinations we will need to account for. This is not only relevant to chunks themselves, but also to the information they encode. Take, for instance, a term definition that relies on other terminology that has been captured. In our effort to reduce duplication and maintain a single source of information, we intend to be able to create chunks with references to other chunks such that the definition can use the known terminology.

To define a term with references to other terms, we need to ensure we are creating our definitions by projecting relevant knowledge into the definition, but also ensuring we have the flexibility to extend that knowledge. With that in mind, we created a Domain Specific Language (DSL) for creating and combining (English language) sentences aptly called `Sentence`. In its most basic form, `Sentence` will simply wrap

```
acceleration = dccWDS "acceleration" (cn' "acceleration")
  (S "the rate of change of a body's" +:+ phrase velocity)
position = dcc "position" (cn' "position")
  "an object's location relative to a reference point"
velocity = dccWDS "velocity" (cnIES "velocity")
  (S "the rate of change of a body's" +:+ phrase position)
```

Figure 4.3: Projecting knowledge into a chunk's definition

a string. However, by defining a number of helper functions and other useful utilities, our `Sentence` DSL can be used to combine, change case, pluralize, and more. A simple example of a series of chunks that utilize the `Sentence` DSL to derive their definitions can be seen in Figure 4.3. We use the `phrase` helper function to pull the appropriate sentence out of the `velocity` chunk and the combinator (`+:+`) to concatenate the sentences. Note that position uses a different smart constructor (for non-derived definitions) and so doesn't require the sentence constructor (`S`) around its definition.

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 Iteration and refinement

[TODO: Figure out what belongs here and what belongs in results —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.

- Tau example (see issue 348) and its implications - symbols and definitions didn't match. -¿ implicit 1m depth into the page (means we may need to change the equations). Resistive and mobilizing shear switched throughout the original docs – impossible with Drasil.

- [This next one might belong in future work —DS] Implicit assumptions -¿ Issue 91. We take for granted things are "physical materials", but this is an assumption that could be codified and made explicit to the system (which would allow us some more flexibility).

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):

```
1 def func_NFL(inParams, q_hat_tol):
2     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.) [Kolmogorov complexity? —DS]

- One major thing to point out here: The most common issues we ran into while attempting to generate our versions of the case study softifacts was that of implicit knowledge that was typically assumed to be "understood" in the context of the domain (i.e. domain experts have tacit knowledge and there are many undocumented assumptions).

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 Mundane Value

[Title should change since some of these may not be so "mundane", but I want to cover a few very specific results and they don't warrant their own sections —DS].

- Consistency by construction

- No undefined symbols - use Tau example?

- "Free" sections - Ref mats can all be generated from "system information" with little effort, unlike manual creation.

5.5 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

- Will allow for much more in-depth sanity checking of generated softifacts

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.

[Developer experience plugins to improve usability would also be useful. Something like a VS Code extension —DS].

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:

- Test the difficulty of create net new software in Drasil vs traditional methods
- Test the difficulty of finding and removing inconsistencies/errors using Drasil vs traditional methods
- Comparing costs (incl. developer time) of maintenance using Drasil vs traditional methods

[More TBD? —DS]

[Might want to rephrase the above into "We want to create a research program to answer the following questions" and then list out the questions we want to answer, namely "Is it easier to create software when using Drasil?", "Is it easier to find and correct errors when using Drasil?", and "Are there significant time savings in long-term maintenance when using Drasil?" —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

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
A	B	C	D
A	B	C	D

Bibliography

[1] Karen S. Ackroyd, Steve H. Kinder, Geoff R. Mant, Mike C. Miller, Christine A. Ramsdale, and Paul C. Stephenson. 2008. Scientific Software Development at a Research Facility. *IEEE Software* 25, 4 (July/August 2008), 44–51.

[2] Shereef Abu Al-Maati and Abdul Aziz Boujarwah. 2002. Literate software development. *Journal of Computing Sciences in Colleges* 18, 2 (2002), 278–289.

[3] Jan Bosch. 2000. *Design and use of software architectures: adopting and evolving a product-line approach*. Addison-Wesley.

[4] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. 2007. Software Development Environments for Scientific and Engineering Software: A Series of Case Studies. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 550–559. `https://doi.org/10.1109/ICSE.2007.77`

[5] Krzysztof Czarnecki and Ulrich W Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional.

[6] Michael Deck. 1996. Cleanroom and object-oriented software engineering: A

unique synergy. In *Proceedings of the Eighth Annual Software Technology Conference, Salt Lake City, USA*.

[7] Steve M. Easterbrook and Timothy C. Johns. 2009. Engineering the Software for Understanding Climate Change. *Comuting in Science & Engineering* 11, 6 (November/December 2009), 65–74. https://doi.org/10.1109/MCSE.2009.193

[8] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. 2009. Scribble: Closing the book on ad hoc documentation tools. In *ACM Sigplan Notices*, Vol. 44. ACM, 109–120.

[9] Peter Fritzson, Johan Gunnarsson, and Mats Jirstrand. 2002. MathModelica-An extensible modeling and simulation environment with integrated graphics and literate programming. In *2nd International Modelica Conference, March 18-19, Munich, Germany*.

[10] Robert Gentleman and Duncan Temple Lang. 2012. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics* (2012).

[11] Marco S Hyman. 1990. Literate C++. *COMP. LANG.* 7, 7 (1990), 67–82.

[12] Cezar Ionescu and Patrik Jansson. 2012. Dependently-Typed Programming in Scientific Computing — Examples from Economic Modelling. In *Revised Selected Papers of the 24th International Symposium on Implementation and Application of Functional Languages (Lecture Notes in Computer Science)*, Vol. 8241. Springer International Publishing, 140–156. https://doi.org/10.1007/978-3-642-41582-1_9

[13] Andrew Johnson and Brad Johnson. 1997. Literate Programming Using `Noweb`. *Linux Journal* 42 (October 1997), 64–69.

[14] Diane Kelly. 2013. Industrial Scientific Software: A Set of Interviews on Software Development. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '13)*. IBM Corp., Riverton, NJ, USA, 299–310. `http://dl.acm.org/citation.cfm?id=2555523.2555555`

[15] Diane Kelly. 2015. Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. *Journal of Systems and Software* 109 (2015), 50–61. `https://doi.org/10.1016/j.jss.2015.07.027`

[16] Diane F. Kelly. 2007. A Software Chasm: Software Engineering and Scientific Computing. *IEEE Softw.* 24, 6 (2007), 120–119. `https://doi.org/10.1109/MS.2007.155`

[17] D. 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

[18] Jeffrey Kotula. 2000. Source code documentation: an engineering deliverable. In *tools*. IEEE, 505.

[19] Friedrich Leisch. 2002. Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat*. Springer, 575–580.

[20] Russell V Lenth. 2009. StatWeave users manual. *URL http://www. stat. uiowa. edu/˜ rlenth/StatWeave* (2009).

[21] Russell V Lenth, Søren Højsgaard, et al. 2007. SASweave: Literate programming using SAS. *Journal of Statistical Software* 19, 8 (2007), 1–20.

[22] Harlan D Mills, Richard C Linger, and Alan R Hevner. 1986. Principles of information systems analysis and design. (1986).

[23] Nedialko S. Nedialkov. 2006. *VNODE-LP — A Validated Solver for Initial Value Problems in Ordinary Differential Equations*. Technical Report CAS-06-06-NN. Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 4K1.

[24] James Milne Neighbors. 1980. *Software construction using components*. University of California, Irvine.

[25] James M Neighbors. 1984. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering* 5 (1984), 564–574.

[26] James M Neighbors. 1989. Draco: A method for engineering reusable software systems. *Software reusability* 1 (1989), 295–319.

[27] Object Management Group. 2000. *The Common Object Request Broker Architecture and Specification*. Technical Report. Object Management Group. `https://www.omg.org/spec/CORBA/2.6/`

[28] Oracle. Accessed February 2023. JavaBeans Component Architecture. `https://docs.oracle.com/javase/8/docs/technotes/guides/beans/index.html`. (Accessed February 2023).

[29] David Lorge Parnas. 2010. Precise Documentation: The Key to Better Software. In *The Future of Software Engineering.* 125–148. https://doi.org/10.1007/978-3-642-15187-3_8

[30] David L. Parnas and P.C. Clements. February 1986. A Rational Design Process: How and Why to Fake it. *IEEE Transactions on Software Engineering* 12, 2 (February 1986), 251–257.

[31] Shari Lawrence Pfleeger and Joanne M. Atlee. 2010. *Software Engineering: Theory and Practice* (4th ed.). Prentice Hall, New Jersey, United States, Chapter 2.

[32] Matt Pharr and Greg Humphreys. 2004. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[33] Vreda Pieterse, Derrick G. Kourie, and Andrew Boake. 2004. A Case for Contemporary Literate Programming. In *Proceedings of the 2004 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT '04).* South African Institute for Computer Scientists and Information Technologists, Republic of South Africa, 2–9. http://dl.acm.org/citation.cfm?id=1035053.1035054

[34] Klaus Pohl, Günter Böckle, and Frank J Linden. 2005. *Software product line engineering: foundations, principles, and techniques.* Springer.

[35] Norman Ramsey. 1994. Literate programming simplified. *IEEE software* 11, 5 (1994), 97.

[36] 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.

[37] Judith Segal. 2005. When Software Engineers Met Research Scientists: A Case Study. *Empirical Software Engineering* 10, 4 (October 2005), 517–536. `https://doi.org/10.1007/s10664-005-3865-y`

[38] Stephen Shum and Curtis Cook. 1993. AOPS: an abstraction-oriented programming system for literate programming. *Software Engineering Journal* 8, 3 (1993), 113–120.

[39] Volker Simonis. 2001. ProgDoc–A Program Documentation System. *Lecture Notes in Computer Science* 2890 (2001), 9–12.

[40] Walid Taha. 2006. A Gentle Introduction to Generative Programming. *ACM Computing Surveys (CSUR)* 37, 4 (2006), 1–28.

[41] Harold Thimbleby. 1986. Experiences of 'Literate Programming'using cweb (a variant of Knuth's WEB). *Comput. J.* 29, 3 (1986), 201–211.

[42] Ye Wu, Dai Pan, and Mei-Hwa Chen. 2001. Techniques for testing component-based software. In *Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems.* 222–232. `https://doi.org/10.1109/ICECCS.2001.930181`