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, 65

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
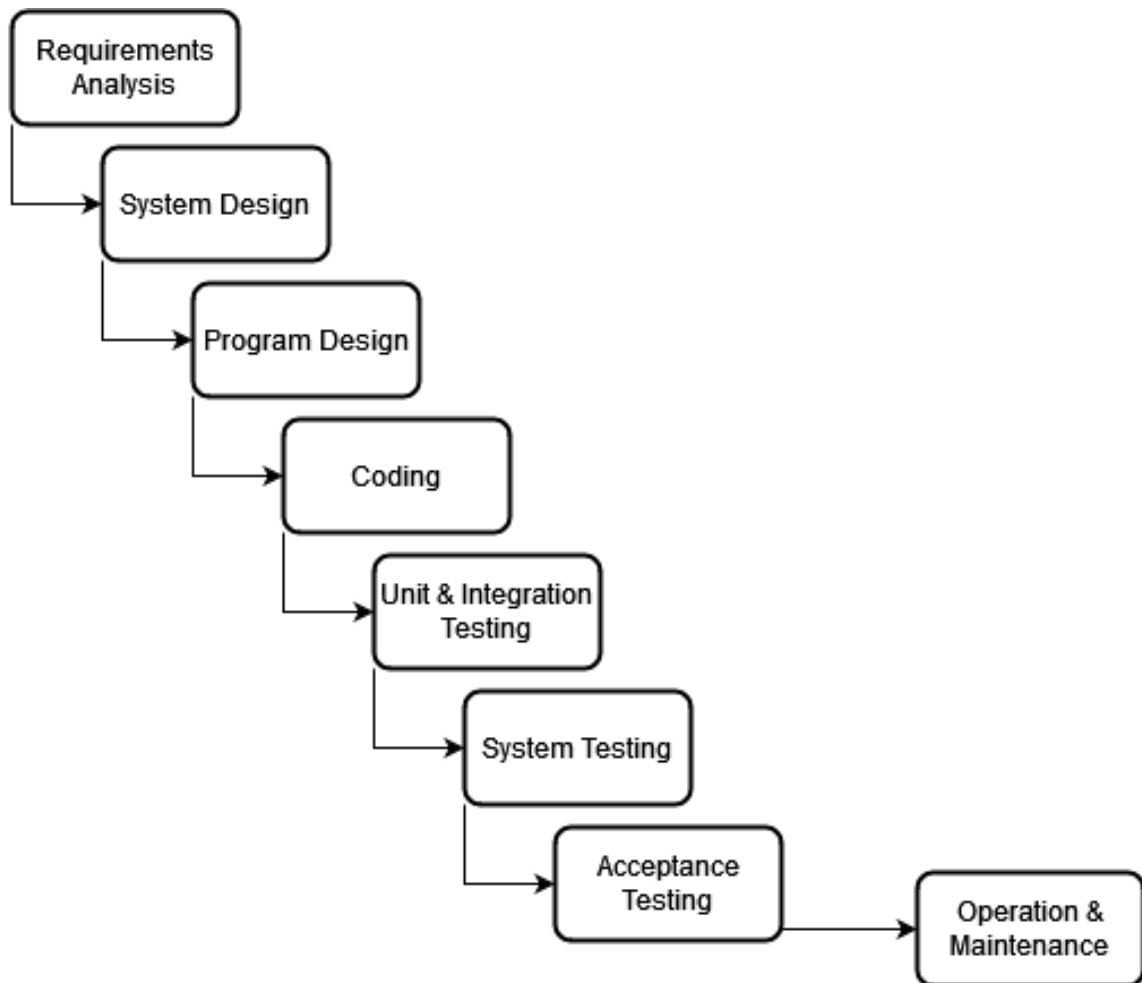
Figure 2.1: The Waterfall Model of Software Development

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.

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.
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.

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]

Chapter 3

A look under the hood:

Our process

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

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

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

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 re-modeling the system.

Name: **SSP**

Problem being solved: A slope of geological mass, composed of soil and rock and sometimes water, is subject to the influence of gravity on the mass. This can cause instability in the form of soil or rock movement which can be hazardous. A system is needed to evaluate the factor of safety of a slope's slip surface and identify the critical slip surface of the slope, as well as the interslice normal force and shear force along the critical slip surface.

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.

---

[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 3.6 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, 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 3.6) we can see that, barring the table values themselves, they are almost identical. The Table of Symbols is simply a table of values, akin to a glossary, specific to the symbols that appear throughout the rest of the document. For each of those symbols, we see the symbol itself, a brief description of what that symbol represents, and the units it is measured in, if applicable. Similarly, the Table of Units lists the Système International d'Unités (SI) Units used throughout the document, their descriptions, and the SI name. Finally,

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

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 above, we can see distinct differences in the types of information contained in each

---

[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

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 same general style: it is a list of modules which represent the leaves of the module

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

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 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 as it serves as the set of instructions that the 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 solution criteria, but would never solve the problem.

As the source code is the executable set of instructions for a system to solve a problem, 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 source code follows almost identically in structure to the module decomposition. The

```
/src/Python
├── Calc.py
├── Constants.py
├── ContoursADT.py
├── Control.py
├── Exceptions.py
├── FunctADT.py
├── GlassTypeADT.py
├── Input.py
├── LoadASTM.py
├── Output.py
├── SeqServices.py
└── ThicknessADT.py
```
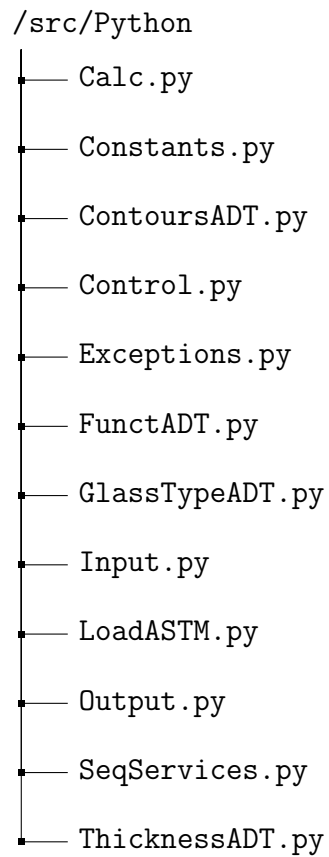
Figure 3.5: Python source code directory structure for GlassBR

only difference is that we also observe an exceptions module defining the different varieties of exceptions that will be thrown by the other modules. While this is a fairly trivial difference, likely made for ease of maintenance, readability, and extensibility, it highlights this fairly obvious difference between the two softifacts. A difference which we speculate was caused by a change made during the implementation phase where the MG was not updated to reflect the addition of an exceptions module.

Looking deeper into the code, as shown in Figure 3.7 we see

3.3 Patterns and repetition and patterns and repetition – (OR – Repeating patterns and patterns that repeat –)

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

[This part isn't exactly true with the reorg of info. I'm going to show some patterns here instead —DS]. From the above sections, we see many emerging patterns in our softifacts. Ignoring, for now, the organizational patterns from the Smith et al. templates we can already see simple patterns emerging.

Returning to our example from Section 3.2, looking only at the reference section of our SRS template, we have already found three subsections that contain the majority of their information in the same organizational structure: a table of symbols and general information relevant to those symbols. Additionally, we can see that the Table of Units and Table of Symbols have an introductory blurb preceding the tables

```python
 4  import math
 5
 6  import Interpolation
 7
 8  ## \brief Calculates stress distribution factor (Function) based on Pbtol
 9  # \param inParams structure holding the input values
10  # \return stress distribution factor (Function) based on Pbtol
11  def func_J_tol(inParams):
12      return math.log(math.log(1.0 / (1.0 - inParams.P_btol)) * ((inParams.a * inParams
        .b) ** (7.0 - 1.0) / (2.86e-53 * (7.17e10 * inParams.h ** 2.0) ** 7.0 * inParams.
        LDF)))
13
14  ## \brief Calculates applied load (demand): 3 second duration equivalent pressure (Pa
        )
15  # \param inParams structure holding the input values
16  # \return applied load (demand): 3 second duration equivalent pressure (Pa)
17  def func_q(inParams):
18      return Interpolation.interpY("TSD.txt", inParams.SD, inParams.w_TNT)
19
20  ## \brief Calculates dimensionless load
21  # \param inParams structure holding the input values
22  # \param q applied load (demand): 3 second duration equivalent pressure (Pa)
23  # \return dimensionless load
24  def func_q_hat(inParams, q):
25      return q * (inParams.a * inParams.b) ** 2.0 / (7.17e10 * inParams.h ** 4.0 *
        inParams.GTF)
26
27  ## \brief Calculates tolerable load
28  # \param inParams structure holding the input values
29  # \param J_tol stress distribution factor (Function) based on Pbtol
30  # \return tolerable load
31  def func_q_hat_tol(inParams, J_tol):
32      return Interpolation.interpY("SDF.txt", inParams.AR, J_tol)
33
34  ## \brief Calculates stress distribution factor (Function)
35  # \param inParams structure holding the input values
36  # \param q_hat dimensionless load
37  # \return stress distribution factor (Function)
38  def func_J(inParams, q_hat):
39      return Interpolation.interpZ("SDF.txt", inParams.AR, q_hat)
40
41  ## \brief Calculates non-factored load: three second duration uniform load associated
         with a probability of breakage less than or equal to 8 lites per 1000 for
        monolithic AN glass (Pa)
42  # \param inParams structure holding the input values
43  # \param q_hat_tol tolerable load
44  # \return non-factored load: three second duration uniform load associated with a
        probability of breakage less than or equal to 8 lites per 1000 for monolithic AN
        glass (Pa)
45  def func_NFL(inParams, q_hat_tol):
46      return q_hat_tol * 7.17e10 * inParams.h ** 4.0 / (inParams.a * inParams.b) ** 2.0
```

```python
48  ## \brief Calculates risk of failure
49  # \param inParams structure holding the input values
50  # \param J stress distribution factor (Function)
51  # \return risk of failure
52  def func_B(inParams, J):
53      return 2.86e-53 / (inParams.a * inParams.b) ** (7.0 - 1.0) * (7.17e10 * inParams.
        h ** 2.0) ** 7.0 * inParams.LDF * math.exp(J)
54
55  ## \brief Calculates load resistance: the uniform lateral load that a glass
        construction can sustain based upon a given probability of breakage and load
        duration as defined in (pp. 1 and 53) Ref: astm2009 (Pa)
56  # \param inParams structure holding the input values
57  # \param NFL non-factored load: three second duration uniform load associated with a
        probability of breakage less than or equal to 8 lites per 1000 for monolithic AN
        glass (Pa)
58  # \return load resistance: the uniform lateral load that a glass construction can
        sustain based upon a given probability of breakage and load duration as defined
        in (pp. 1 and 53) Ref: astm2009 (Pa)
59  def func_LR(inParams, NFL):
60      return NFL * inParams.GTF * 1.0
61
62  ## \brief Calculates 3 second load equivalent resistance safety requirement
63  # \param LR load resistance: the uniform lateral load that a glass construction can
        sustain based upon a given probability of breakage and load duration as defined
        in (pp. 1 and 53) Ref: astm2009 (Pa)
64  # \param q applied load (demand): 3 second duration equivalent pressure (Pa)
65  # \return 3 second load equivalent resistance safety requirement
66  def func_isSafeLR(LR, q):
67      return LR > q
68
69  ## \brief Calculates probability of breakage: the fraction of glass lites or plies
        that would break at the first occurrence of a specified load and duration,
        typically expressed in lites per 1000 (Ref: astm2016)
70  # \param B risk of failure
71  # \return probability of breakage: the fraction of glass lites or plies that would
        break at the first occurrence of a specified load and duration, typically
        expressed in lites per 1000 (Ref: astm2016)
72  def func_P_b(B):
73      return 1.0 - math.exp(-B)
74
75  ## \brief Calculates probability of glass breakage safety requirement
76  # \param inParams structure holding the input values
77  # \param P_b probability of breakage: the fraction of glass lites or plies that would
         break at the first occurrence of a specified load and duration, typically
        expressed in lites per 1000 (Ref: astm2016)
78  # \return probability of glass breakage safety requirement
79  def func_isSafePb(inParams, P_b):
80      return P_b < inParams.P_btol
```

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

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

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

themselves, whereas the Table of Abbreviations and Acronyms does not. Inspecting across case studies, we can see that the introduction to the Table of Units is simply boilerplate text dropped into each case study verbatim, thus is completely generic and applicable to *any* software system using SI units. The introduction to the Table of Symbols appears to be boilerplate across several examples, however, it does have some variation which we can see by comparing Figure 3.6b to Figure 3.8 (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 system-specific knowledge being encoded. We can intuit this conclusion based solely on each system solving a different problem, however, we have confirmed this by solely looking at the structure (patterns) of one section of the SRS.

The reference section of the SRS provides a lot of knowledge in a very straightforward and organized manner. The basic units provided in the table of units give a

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

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

- inter-project (repetition throughout different views + other patterns.) vs intra-project knowledge (repetition across projects/family members, minor modifications, but fundamentally the same + other patterns.) - Hint at chunkifying/parceling out the fundamental (system/view-agnostic) knowledge vs the specific knowledge

[Need to talk about knowledge projection/transformation in this section - Use a relevant example from Case studies, maybe from PCM? Something like $\Delta U =$

$Q - W$ is the same as "Total energy within a closed system must be conserved" but transformed for / projects out what is relevant to a given audience. - This will be useful for the following section —DS]

3.4   Organizing knowledge - a fluid approach

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

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

3.5   The seeds of Drasil

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

Chapter 4

Drasil

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

4.1   What Drasil is and isn't

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

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

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

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

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

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

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

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

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

4.2   Our Ingredients: Organizing and Capturing Knowledge

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

4.3   Recipes: Codifying Structure

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

4.4   Cooking it all up: Generation/Rendering

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

4.5   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:

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

and its code representation (in python):

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

At a glance, are the code and formula equivalent?

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

5.2   Originals vs. Reimplementations

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

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.

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 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

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`