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:	xvi, 119

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

Lay Abstract **iii**

Abstract **iv**

Acknowledgements **vi**

List of Figures **xii**

List of Tables **xiii**

Notation, Definitions, and Abbreviations **xiv**

Declaration of Academic Achievement **xvi**

1 Introduction **1**

1.1 Value in the mundane 3

1.2 Problem Statement and Motivation 3

1.3 Scope 4

1.4 Roadmap 5

1.5 Contributions 5

2 Background **8**

2.1 Software Artifacts 8

2.2 Software Reuse and Software Families 14

2.3 Literate Approaches to Software Development 19

2.4 Generative Programming 23

3 A look under the hood:

Our process **25**

3.1 A (very) brief introduction to our case study systems 26

3.2 Breaking down softifacts 30

3.3 Identifying Repetitive Redundancy 42

3.4 Organizing knowledge - a fluid approach 49

3.5 Summary - The seeds of Drasil 52

4 Drasil **54**

4.1 Drasil in a nutshell 54

4.2 Our Ingredients: Organizing and Capturing Knowledge 56

4.3 Recipes: Codifying Structure 68

4.4 Cooking it all up: Generation/Rendering 83

4.5 Iteration and refinement 94

4.6 Summary 97

5 Results **99**

5.1 Mundane Value 99

5.2 “Pervasive” Bugs 103

5.3 Originals vs. Reimplementations 104

5.4 Design for change 104

5.5 Usability 105

6 Future Work **106**

6.1 Inference 106

6.2 Typed Expression Language 107

6.3 Model types 107

6.4 Usability 107

6.5 Variants 107

6.6 Many more artifacts/more document knowledge 108

6.7 More display variabilities 108

6.8 More languages (?) 108

6.9 More scientific knowledge 108

6.10 More computational knowledge 108

6.11 Measuring Drasil's Effectiveness 109

7 Conclusion **110**

A Your Appendix **111**

B Long Tables **112**

List of Figures

2.1 The Waterfall Model of Software Development 9

3.1 The Table of Contents from the ([expanded? —DS]) Smith et al. template 31

3.2 The reference sections of GlassBR 33

3.3 Table of Contents for GlassBR Module Guide 36

3.4 Calc Module from the GlassBR Module Guide 37

3.5 Python source code directory structure for GlassBR 39

3.6 Source code of the Calc.py module for GlassBR 41

3.7 Table of Symbols (truncated) Section from GamePhysics 43

3.8 Data Definition for Dimensionless Load ($\hat{q}$) from GlassBR SRS 44

3.9 Theoretical Model of conservation of thermal energy found in both the SWHS and NoPCM SRS 47

3.10 Instance Model difference between SWHS and NoPCM 48

4.1 NamedChunk Definition 57

4.2 Some example instances of `ConceptChunk` using the `dcc` smart constructor 58

4.3 Projecting knowledge into a chunk's definition 60

4.4 The force `ConceptChunk` 61

4.5 The force and acceleration `UnitalChunks` 61
4.6 The fundamental SI Units as Captured in Drasil 62
4.7 Examples of chunks for units derived from other SI Unit chunks . . . 62
4.8 Defining real number multiplication in `Expr` 64
4.9 Encoding Newton's second law of motion 64
4.10 Newton's second law of motion as a *Chunk* 65
4.11 A subset of Drasil's Chunk Classification system showing some of the encoded property relationships 67
4.12 A subset of Drasil's chunks showing how different chunks may belong to multiple classifications 68
4.13 Recipe Language for SRS 70
4.14 `ReqrmntSec` Decomposition and Definitions 71
4.15 SRS Recipe for GlassBR 72
4.16 System Information for GlassBR 73
4.17 The `iMods` definition 75
4.18 The Recipe for GlassBR's Executable Code 77
4.19 The Recipe Language for Executable Code (CodeSpec) 78
4.20 The `readTableMod` module definition 80
4.21 The `Choices` object definition 80
4.22 `genTeX` function for LaTeX rendering 85
4.23 generateCode and genPackage for code rendering 86
4.24 `prntDoc'` function for document output 87
4.25 `prntMake` function for document output 87
4.26 createCodeFiles for code output 88

4.27 Definition of $\hat{q}$ as a chunk in the GlassBR knowledge base 89
4.28 Raw generated LaTeXof the $\hat{q}$ Data Definition 91
4.29 $\hat{q}$ as defined in the generated python code 93

List of Tables

2.1 Comparison of Rational Design Process and Waterfall Model 12

2.2 A summary of the Audience for each of the most common softifacts and what problem that softifact is solving 15

4.1 System information object breakdown - every property is represented by chunks encoding given information 74

4.2 CodeSpec object breakdown 79

4.3 Choices object breakdown 81

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]

[Give examples of what audience writing for - know classical software engineering —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.

[If we can truly understand the differences and commonalities between software artifacts, we can develop tools to capitalize on them and streamline the process of creating and maintaining software artifacts. For that we first need to break down common artifacts to develop an understanding of their contents, intended audience, and how they differ whether on intra- or inter-project scales —DS].

We propose a new framework, Drasil, alongside a knowledge-centric view of software, to help take advantage of inherent redundancy, while avoiding [/reducing —DS] 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

?? [What we're doing is not big and flashy, instead it is focused on the tedium we deal with in our day-to-day lives as Software Engineers. We are focused on improving the development experience holistically. The small annoyances can be more frustrating than solving larger problems. Ex. The feeling of hours spent banging heads against walls for an off-by-one or a typo that didn't get caught, versus the sublime joy of finding a novel, interesting, scalable, and maintainable solution to a problem. —DS]

[More about the little pain points should go here - i.e. having to go manually update artifacts after modifying code, particularly in regulated industries —DS]

1.2 Problem Statement and Motivation

Despite the recognized value of documentation and software artifacts (softifacts), the process of creating and maintaining them is fraught with redundancy, tedium, and potential risks of inconsistency. As software evolves, artifacts (requirements documents, design documents, source code, etc.) often fall out of sync, leading to errors, increased maintenance costs, and reduced trust in documentation. This thesis addresses the challenge: *How can we systematically reduce unnecessary redundancy and improve consistency across software artifacts, especially in domains where correctness and traceability are critical?*

1.3 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.

1.4 Roadmap

The remainder of this thesis is organized as follows:

- **Chapter 2: Background** reviews the literature on software artifacts, reuse, literate programming, and generative programming.
- **Chapter 3: Our Process** details the methods used to analyze artifacts and identify redundancy.
- **Chapter 4: The Drasil Framework** presents the design and implementation of Drasil.
- **Chapter 5: Results** discusses the outcomes of applying Drasil to case studies.
- **Chapter 6: Future Work** outlines potential extensions and open questions.
- **Chapter 7: Conclusion** summarizes the findings and their implications.

1.5 Contributions

The main contributions of this thesis are:

- A systematic analysis of redundancy and inconsistency in software artifacts, with a focus on scientific computing.
- The design and implementation of the Drasil framework, which enables knowledge-centric generation of multiple software artifacts from a single source of truth.
- A demonstration of Drasil's effectiveness through reimplementation of several case studies, highlighting improvements in consistency, maintainability, and traceability.
- A discussion of the limitations, challenges, and future directions for knowledge-centric software artifact generation.

[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

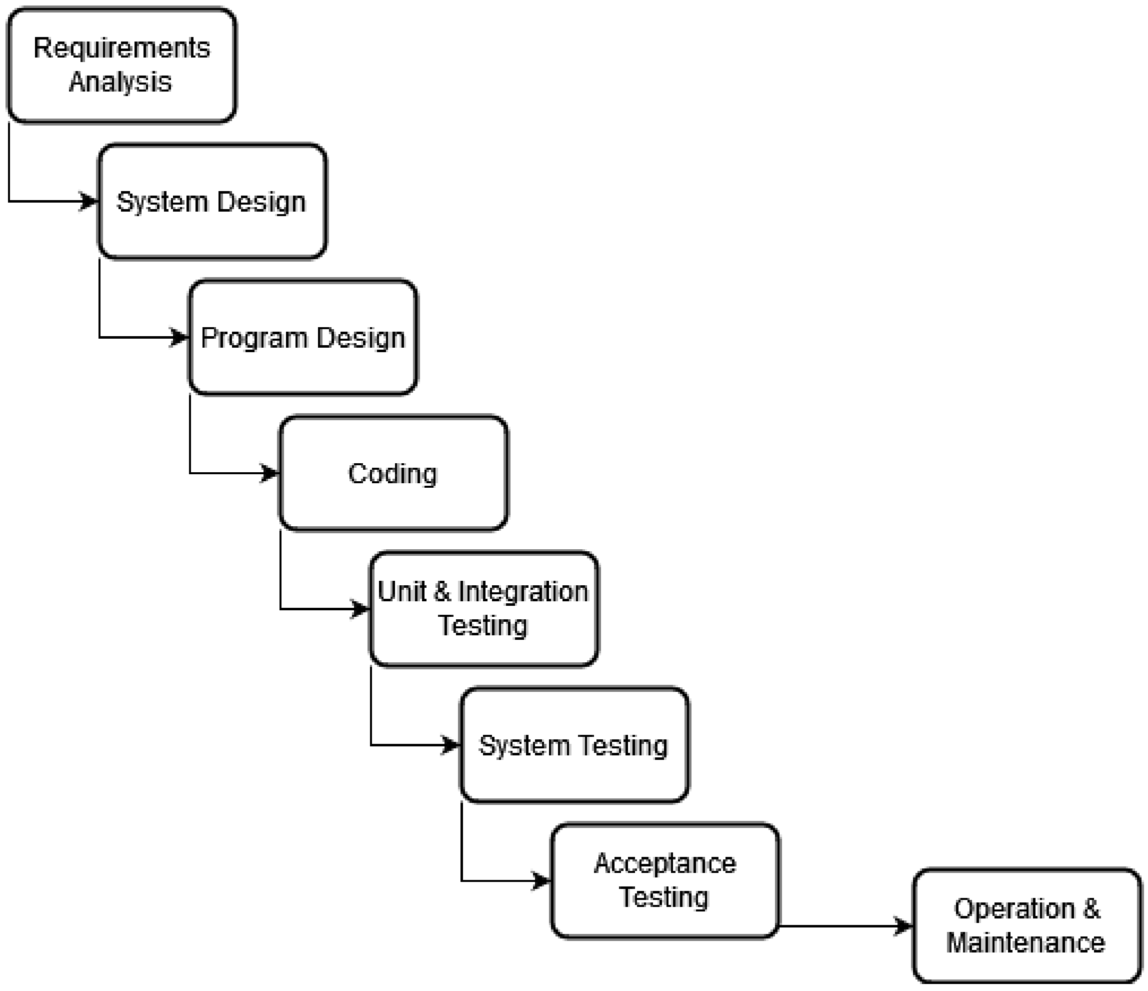


Figure 2.1: The Waterfall Model of Software Development

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

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.

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”. Draco was one of the earliest and most influential frameworks for software reuse, particularly in the context of domain-specific software generation. The core idea behind Draco is to enable the construction of software systems by composing reusable, domain-specific components called ”domain knowledge modules.” In Draco, a software engineer first defines a domain model, capturing the essential abstractions and operations relevant to a particular problem area. The system then provides a set of reusable transformations and code generation templates tailored to that domain. By selecting and configuring these modules, developers can automatically generate large portions of a software system, significantly reducing manual coding effort. Draco’s approach anticipated many later developments in software product lines and generative programming, emphasizing the importance of capturing domain knowledge explicitly and supporting its reuse across multiple projects. Although Draco itself is no longer widely used, its concepts have influenced modern frameworks for generative programming and domain-specific language design [25, 5].

[TODO: refine the above and move citations as needed]

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. Actually might belong in the next chapter - iteration and refinement —DS]

[Add something about how this is about deving the framework itself, not dev in general. This is about our developing of the framework itself —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.

[Add an example, something like hghc or F=ma, to set the scene —DS].

[Make sure to tell the audience why information is being presented here and how it's relevant to the chapter ahead. Give an outline of the entire chapter, not just the first bit. Get into some detail. —DS]

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 1.3, we have chosen software systems that follow the '*input*' → '*process*' → '*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.

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

3.2.1 SRS

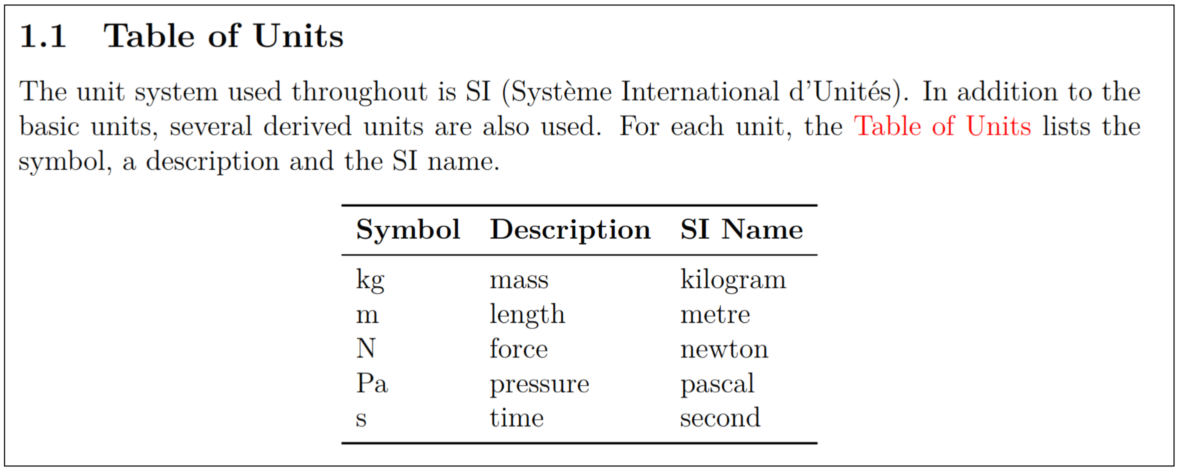
To start, we look at the Software Requirements Specification (SRS). The SRS (or some incarnation of it) is one of the most important artifacts for any software project as it specifies what problem the software is trying to solve. There are many ways to state this problem, and the template from Smith 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

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

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.

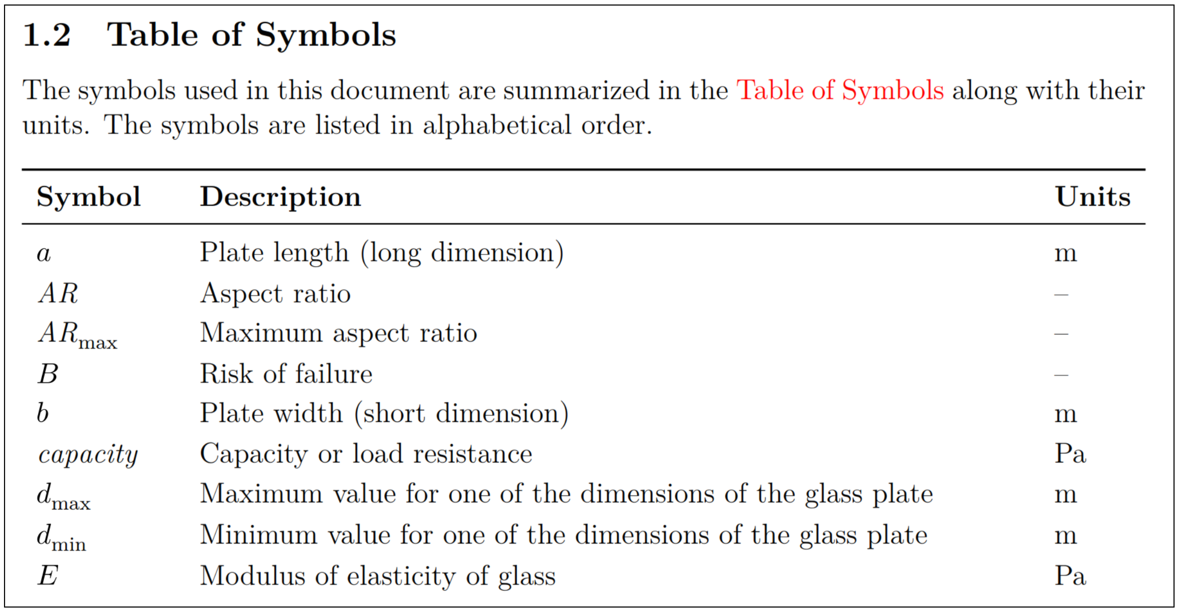
Symbol	Description	SI Name
kg	mass	kilogram
m	length	metre
N	force	newton
Pa	pressure	pascal
s	time	second

(a) Table of Units Section

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) 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.

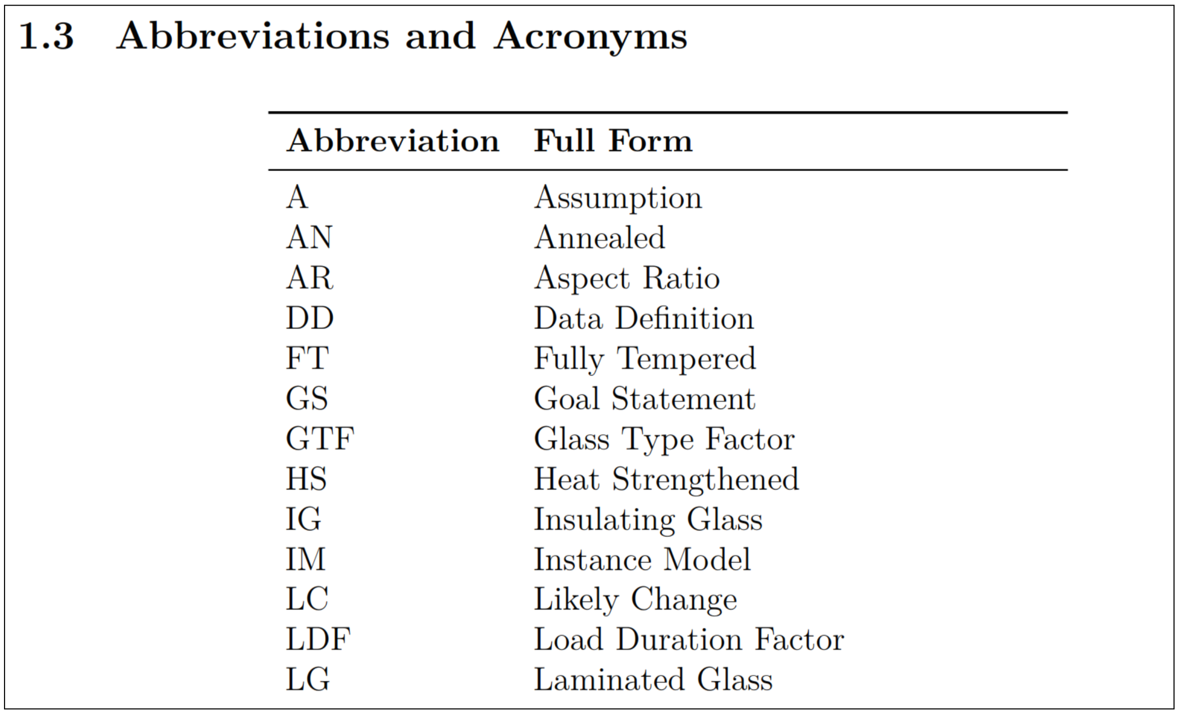


1.2 Table of Symbols

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

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 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

[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.

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]

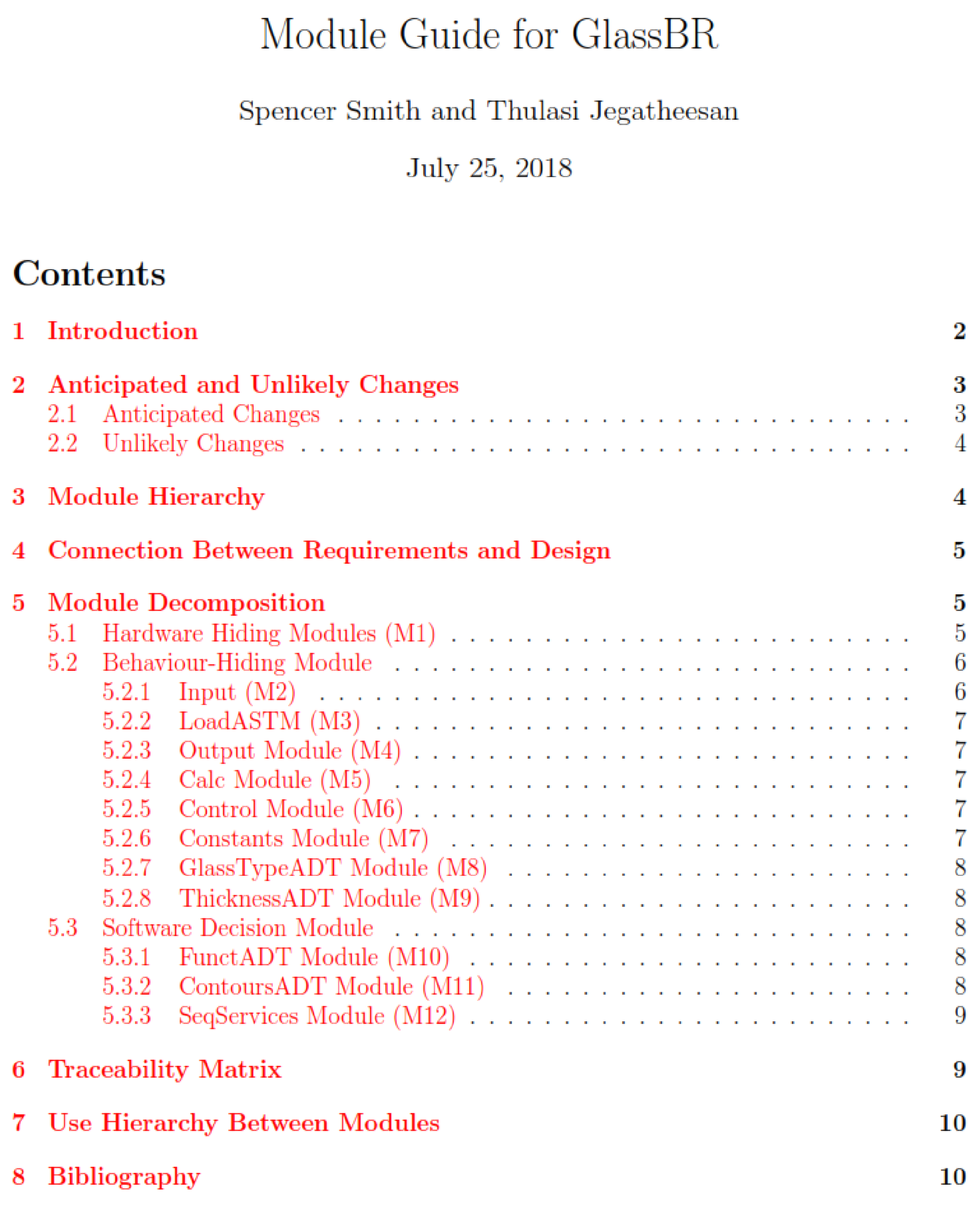
[Example to use should be a DD/IM from GlassBR, goes to calculations module in the MG, and finally a method in the source code —DS]

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



Module Guide for GlassBR

Spencer Smith and Thulasi Jegatheesan

July 25, 2018

Contents

1 Introduction **2**

2 Anticipated and Unlikely Changes **3**
2.1 Anticipated Changes 3
2.2 Unlikely Changes 4

3 Module Hierarchy **4**

4 Connection Between Requirements and Design **5**

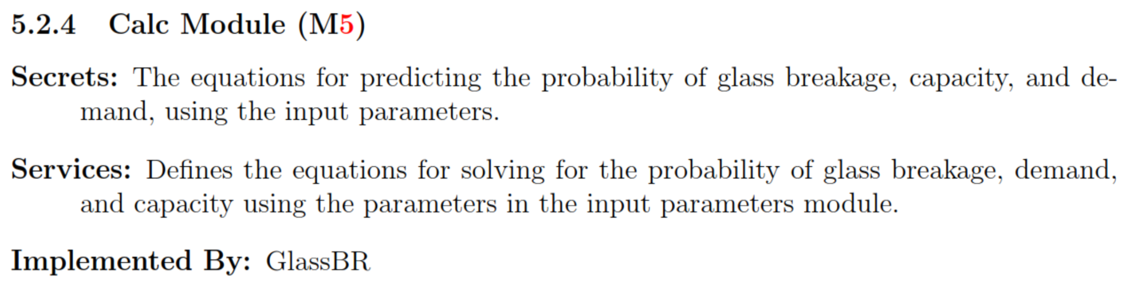
5 Module Decomposition **5**
5.1 Hardware Hiding Modules (M1) 5
5.2 Behaviour-Hiding Module 6
5.2.1 Input (M2) 6
5.2.2 LoadASTM (M3) 7
5.2.3 Output Module (M4) 7
5.2.4 Calc Module (M5) 7
5.2.5 Control Module (M6) 7
5.2.6 Constants Module (M7) 7
5.2.7 GlassTypeADT Module (M8) 8
5.2.8 ThicknessADT Module (M9) 8
5.3 Software Decision Module 8
5.3.1 FunctADT Module (M10) 8
5.3.2 ContoursADT Module (M11) 8
5.3.3 SeqServices Module (M12) 9

6 Traceability Matrix **9**

7 Use Hierarchy Between Modules **10**

8 Bibliography **10**

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

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 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.

```
/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

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 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)

```python
from Input import *
from ContoursADT import *
from math import log, exp

## @brief Calculates the Dimensionless load
#  @return the unitless load
def calc_q_hat( q, params ):
    upper = q * (params.a * params.b) ** 2
    lower = params.E * (params.h ** 4) * params.GTF
    return ( upper / lower )
## @brief Calculates the Stress distribution factor based on Pbtol
#  @return the unitless stress distribution factor
def calc_J_tol( params ):
    upper1 = 1
    lower1 = 1 - params.Pbtol

    upper2 = (params.a * params.b) ** (params.m - 1)
    lower2 = params.k * ((params.E * (params.h ** 2)) ** (params.m)) * params.LDF

    return (log( (log(upper1 / lower1)) * (upper2 / lower2) ))
## @brief Calculates the Probability of glass breakage
#  @return unitless probability of breakage
def calc_Pb( B ):
    output = 1 - (exp(-B))
    if not (0 < output < 1):
        raise InvalidOutput("Invalid output!")
    return (output)
## @brief Calculates the Risk of failure
#  @return unitless risk of failure
def calc_B( J, params ):
    upper = params.k * ((params.E * (params.h) ** 2) ** params.m) * params.LDF * exp(
    J)
    lower = ((params.a * params.b) ** (params.m - 1))
    return ( upper / lower )
## @brief Calculates the Non-factores load
#  @return unitless non-factored load
def calc_NFL( q tol , params ):
    upper = q tol  * params.E * (params.h ** 4)
    lower = (params.a * params.b) ** 2
    return ( upper / lower )
## @brief Calculates the Load resistance
#  @return unitless load resistance
def calc_LR( NFL, params ):
    return ( NFL * params.GTF * params.LSF )
## @brief Calculates Safetey constraint 1
#  @return true if the calculated probability is less than the tolerable probability
def calc_is_safePb( Pb, params ):
    if (Pb < params.Pbtol):
        return True
    else:
        return False
## @brief Calculates Safetey constraint 2
#  @return true if the load resistance is greater than the load
def calc_is_safeLR( LR, q ):
    if (LR > q):
        return True
    else:
        return False
```

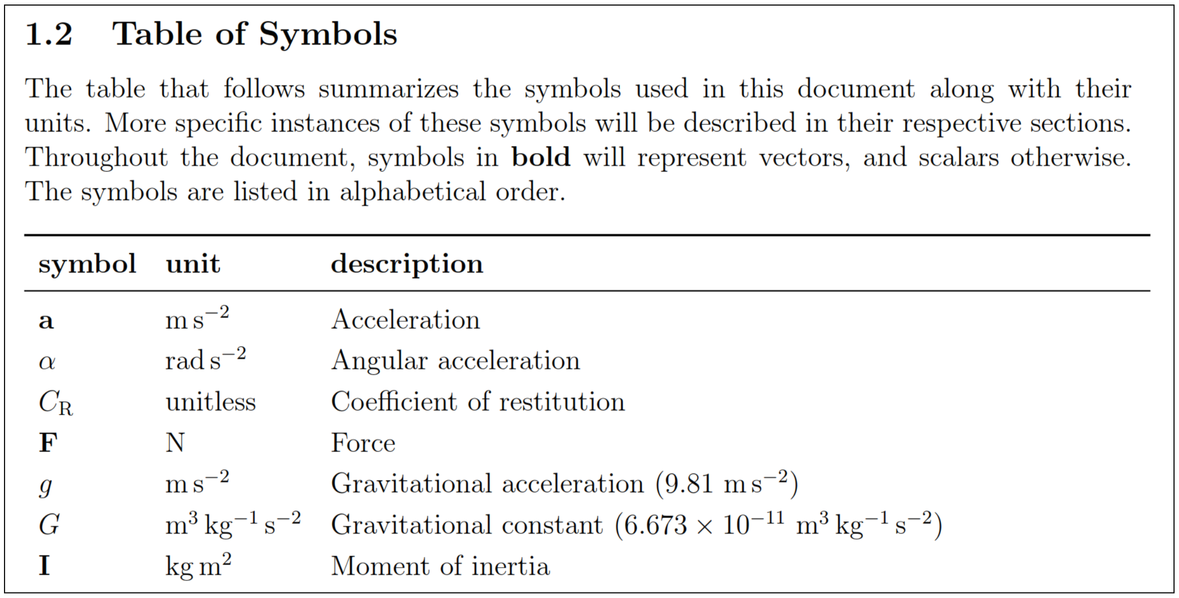
Figure 3.6: Source code of the Calc.py module for GlassBR

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.

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



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
$\mathbf{a}$	$\mathrm{m\,s^{-2}}$	Acceleration
α	$\mathrm{rad\,s^{-2}}$	Angular acceleration
C_{R}	unitless	Coefficient of restitution
$\mathbf{F}$	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 ($6.673 \times 10^{-11}\ \mathrm{m^3\,kg^{-1}\,s^{-2}}$)
$\mathbf{I}$	$\mathrm{kg\,m^2}$	Moment of inertia

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

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 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

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

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 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

[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.

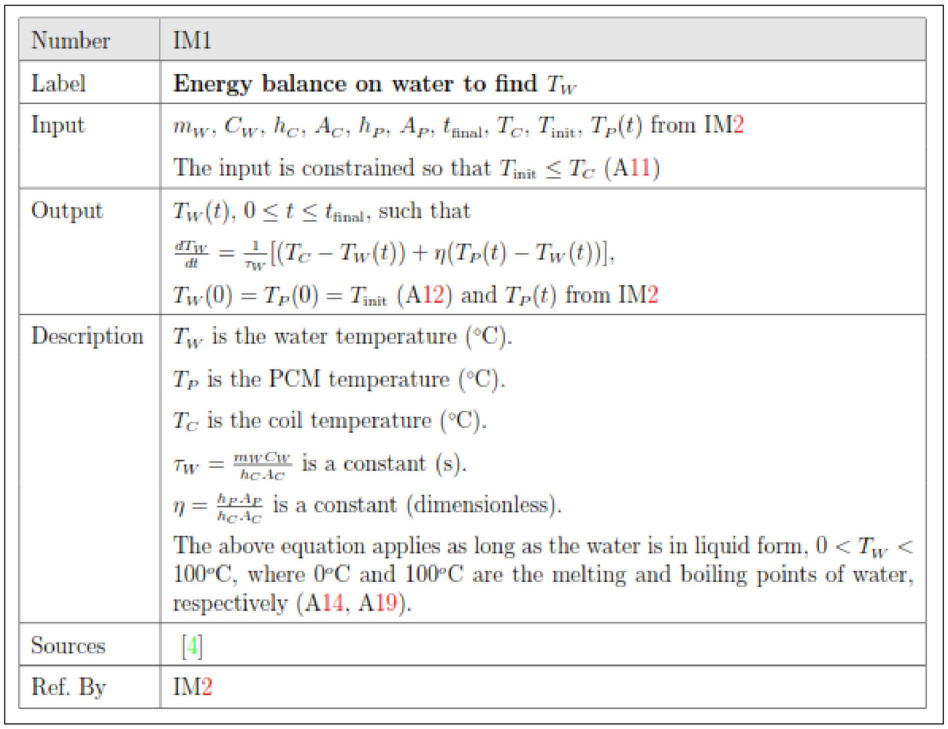
Number	T1
Label	**Conservation of thermal energy**
Equation	$-\nabla \cdot \mathbf{q} + g = \rho C \frac{\partial T}{\partial t}$
Description	The above equation gives the conservation of energy for transient heat transfer in a material of specific heat capacity C ($\mathrm{J\,kg^{-1}\,°C^{-1}}$) and 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

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 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).

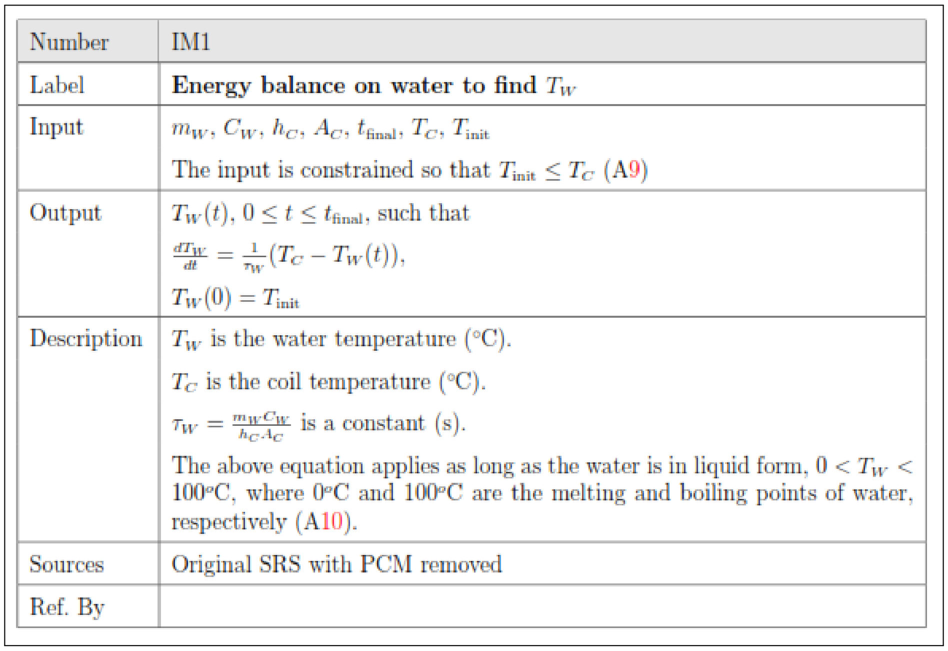
Looking at SWHS and NoPCM, we can easily find identical theoretical models (TMs) as the underlying theory for each system is based on the problem domain (see example in Figure 3.9). However, when we follow the derivations from the TMs to the Instance Models (IMs), we find the resulting equations have 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



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

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 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

[Point out what the reader should have learned from this chapter before anything else —DS]

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

In this chapter, we introduce the Drasil framework and its implementation details, including knowledge-capture mechanisms, the domain specific languages (DSLs) used throughout, and how these components are integrated 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, reflects how our framework spans 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.

Unlike 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 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

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

Figure 4.1: NamedChunk Definition

decided to take an iterative, progressive refinement approach to our knowledge capture mechanisms. This should come as 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.

A single node does not a hierarchy make, but 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

```
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

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 simple instances of `ConceptChunk` 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 Drasil source code or found online at https://jacquescarette.github.io/Drasil/docs/full/

4.2.2 Chunk Combinatorics

[Would we call our 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 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.

~~[Should I work in something about how we use lenses in here, as all complex chunks~~

```
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

~~are instances of the simpler chunks they've been built from? Or is that getting too into the Haskell-specific details?~~ I've written something below —DS]

We also make opportunistic use of composable accessors (commonly known in the Haskell ecosystem as "lenses") to help manage complexity when combining and projecting chunks. Conceptually, these are lightweight, composable getters and setters that let us focus on the logical structure of a chunk (its name, symbol, sentence, expression, units, etc.) without repeatedly writing boilerplate navigation code for nested fields[1]. In practice they let us build derived chunks by projecting the subparts we need, update specific components (for instance a symbol or an attached unit) in a principled, immutable way, and compose transformations cleanly when constructing larger chunks from smaller ones.

Looking back at our examples of the types of knowledge we are interested in from Figure 3.8 we can see that the simple chunks we've defined so far do not even start to cover the full spectrum of our needs. We can see a lot of information missing such as a symbolic representation ($\hat{q}$) and equation ($\hat{q} = \frac{q(ab)^2}{Eh^4\text{GTF}}$). As that is a relatively complex example, we will start by looking at something far simpler: Newton's 2nd Law of motion[**?**] which, roughly translated, states "the net force on a body at any instant of time is equal to the body's acceleration multiplied by its mass or,

[1]Concrete lens usage and examples are available in the code found in the project repository for readers who want the implementation details.

```
force = dcc "force" (cn' "force")
  "an interaction that tends to produce change in the motion of an object"
```

Figure 4.4: The force `ConceptChunk`

```
acceleration = uc CP.acceleration (vec lA) accelU
force = uc CP.force (vec cF) newton
```

Figure 4.5: The force and acceleration `UnitalChunks`

equivalently, the rate at which the body's momentum is changing with time" or as most physics students recognize it $F = ma$ provided we have definitions for F, m, and a.

To understand what exactly is missing, let us look at how we achieve an operational definition of *Force* as a chunk in Drasil. To start, we can see we need a symbolic representation (F) and some way to define an equation. Also, each of force, mass, and acceleration are measured in some form of units (N, kg, and m/s^2 respectively) which we will likely care to capture and keep track of.

We start by building a `ConceptChunk` for the Force concept as seen in Figure 4.4. It is built from a common noun ("force") and a definition. Next we need to capture the units of measurement to the force concept by defining what we have named a `UnitalChunk`. This definition for force can be seen in Figure 4.5. Note that we are not re-defining force in this instance. We are instead using a smart constructor to build atop the existing force chunk (`CP.force`) and adding a symbolic representation (`vec cF`, which is shorthand for the selected vector representation[2] of the capital letter F). We also must capture the units used to measure force (`newton`) which are defined in `Data.Drasil.SI_Units` and can be seen in Figure 4.6. The SI Units are captured using another type of chunk known as a `UnitDefn` which are built atop

[2]More on this in Section 4.3

```
-- * Fundamental SI Units

metre, kilogram, second, kelvin, mole, ampere, candela :: UnitDefn
metre    = fund "metre"    "length"              "m"
kilogram = fund "kilogram" "mass"                "kg"
second   = fund "second"   "time"                "s"
kelvin   = fund "kelvin"   "temperature"         "K"
mole     = fund "mole"     "amount of substance" "mol"
ampere   = fund "ampere"   "electric current"    "A"
candela  = fund "candela"  "luminous intensity"  "cd"
```

Figure 4.6: The fundamental SI Units as Captured in Drasil

```
accelU          = newUnit "acceleration"         $ metre /: s_2
angVelU         = newUnit "angular velocity"     $ radian /: second
angAccelU       = newUnit "angular acceleration" $ radian /: s_2
forcePerMeterU  = newUnit "force per meter"      $ newton /: metre
impulseU        = newUnit "impulse"              $ newton *: second
momtInertU      = newUnit "moment of inertia"    $ kilogram *: m_2
momentOfForceU  = newUnit "moment of force"      $ newton *: metre
springConstU    = newUnit "spring constant"      $ newton /: metre
torqueU         = newUnit "torque"               $ newton *: metre
velU            = newUnit "velocity"             $ metre /: second
```

Figure 4.7: Examples of chunks for units derived from other SI Unit chunks

a `ConceptChunk` and `UnitSymbol`. The `UnitSymbol` is also a chunk that can be one of several other chunk types.[3]. Returning to the force `UnitalChunk`, the smart constructor `uc` we are using will assume we are working in the real number space. There are other smart constructors for other spaces, which are also captured using other chunks elsewhere in Drasil (see: `Language.Drasil.Space`).

Now we approach the terms mass and acceleration in a similar manner. We capture each as a `ConceptChunk` (see Figure 4.3 for a reminder of how we captured acceleration's definition relative to velocity and thus relative to position). Then construct a `UnitalChunk` for each. Acceleration can be seen in Figure 4.5 and mass has a similar `UnitalChunk` that uses the `metre UnitDefn` chunk. More interestingly, the

[3]For brevity we are glossing over many chunk definitions and the differences between them as there are a large variety in use for even simple examples. They are covered in more depth in the full technical documentation found on our github repository.

units for acceleration are derived from other units, and can be seen in Figure 4.7 with some additional derived unit types (taken from `Data.Drasil.Units.Physics`).

Now we have almost everything we need to finish our definition of Newton's second law. Everything thus far has been defined in natural language using our `Sentence` DSL, but it is not well-suited to the one piece we are currently missing: a way to define expressions relating chunks in a universal [language agnostic? —DS] (mathematics) context [/representation —DS].

4.2.3 Relating Chunks via Expressions

Continuing our Newton's Second Law example from Section 4.2.2, we need a means to capture how force is calculated. We know it is calculated relative to mass and acceleration, so we need a way to encode that, preferably in an operational manner. We also know acceleration is itself derived from time and velocity, which is derived from time and position.

When thinking about the types of information we would like to encode with expressions, we have some obvious candidates. We should be able to encode common arithmetic operations (addition, subtraction, multiplication, division, exponentiation, etc.), boolean operations (and, or, not, etc.), comparisons (equal, not equal, less than, greater than), trigonometric functions (sin, tan, cos, arctan, etc.), calculus (derivation, integration, etc.), and vector/matrix operations (dot product, cross product, etc.). For the sake of our example we'll focus on only those we need to define Newton's Second Law of motion: multiplication and derivatives.

As an aside, we will elide details on how we arrived at the current implementation of the expression DSL known as `Expr`, but suffice to say it was driven by a practical,

```
-- | Multiply two expressions (Real numbers).
mulRe l (Lit (Dbl 1))      = l
mulRe (Lit (Dbl 1)) r      = r
mulRe l (Lit (ExactDbl 1)) = l
mulRe (Lit (ExactDbl 1)) r = r
mulRe (AssocA MulRe l) (AssocA MulRe r) = AssocA MulRe (l ++ r)
mulRe (AssocA MulRe l) r = AssocA MulRe (l ++ [r])
mulRe l (AssocA MulRe r) = AssocA MulRe (l : r)
mulRe l r = AssocA MulRe [l, r]
```

Figure 4.8: Defining real number multiplication in `Expr`

```
newtonSLEqn                = sy QPP.mass `mulRe` sy QP.acceleration

velocityEqn, accelerationEqn :: ModelExpr
velocityEqn                = deriv (sy QP.position) QP.time
accelerationEqn            = deriv (sy QP.velocity) QP.time
```

Figure 4.9: Encoding Newton's second law of motion

"lowest common denominator" approach to developing the operations that could be encoded and factoring out commonalities much the same way we did with the rest of the case studies. With that in mind, we have encoded multiplication (for real numbers) as seen in Figure 4.8. `AssocA` refers to an associative arithmetic operator which can be applied across a list of expressions (in this case, `mulRe`, though addition and other associative operations look very similar). Derivation is much simpler to encode, as it is simply a relationship between two existing chunks, ie. `deriv a b` represents taking the derivative of `a` with respect to `b`.

Continuing our example, we see (Figure 4.9) it is fairly trivial to encode the expression for Newton's Second Law using the `Expr` DSL. We are still not done building our Newton's Second Law chunk however, as this specific law should be referable by its own identifier. It is not simply "force", nor the relationship between mass and acceleration, but a combination of both with its own natural language semantics that allow us to refer to it specifically. The full definition for this particular

```
newtonSLQD :: ModelQDef
newtonSLQD = fromEqn' "force" (nounPhraseSP "Newton's second law of motion")
  newtonSLDesc (eqSymb QP.force) Real newtonSLEqn

newtonSLDesc :: Sentence
newtonSLDesc = foldlSent [S "The net", getTandS QP.force, S "on a",
  phrase body `S.is` S "proportional to", getTandS QP.acceleration `S.the_ofThe`
  phrase body `sC` S "where", ch QPP.mass, S "denotes", phrase QPP.mass `S.the_ofThe`
  phrase body, S "as the", phrase constant `S.of_` S "proportionality"]
```

Figure 4.10: Newton's second law of motion as a *Chunk*

chunk can then be found in Figure 4.10. The chunk is defined relative to force and the expression captured in Figure 4.9, alongside a new identifier and description. Notice we are building off of other chunks throughout each piece of the chunk's definition, thus giving us perfect traceability from beginning to end, regardless of whether we are looking at a defining expression or natural language description.

The `Expr` DSL grants us flexibility in defining relationships without imposing any particular structure on a chunk other than "contains an `Expr`", or more specifically "can be modeled using an `Expr`." Our example also shows us that a few simple chunk types can be combined repeatedly to build much more robust, complex, and information-dense chunks. On the other hand, encoding even a relatively simple theory like Newton's Second Law of motion requires us to define chunks down to the fundamentals. Luckily, that is another area where chunks shine as they are infinitely reusable. Once a chunk has been defined and added to our knowledge-base, we never need to redefine it. We simply use it wherever it is needed.

On the topic of using chunks, we now need a means for taking our chunks and structuring them such that we can actually do something useful with them. For example, how would we go from chunks to softifacts? Look forward to that in Section 4.3.

4.2.4 Chunk Classification

[I've modified this section to not only be called something different, but to get into the weeds a bit on chunk classification, so we can show the high level "Class of chunk has this property" instead of "Here are all our chunk datatypes" —DS]

In the previous sections, we used the idea of "chunks" of knowledge as our building blocks. They range in complexity from simple single terms, to complex relationships between expressions and overarching conceptual frameworks. Each chunk is useful for a particular encoding, and complex chunks are built from simpler, more fundamental chunks. With all of the complexities of mixing and matching chunks, we have developed a need for a way to determine which chunks expose the same types of knowledge (and how they differ). Not only would such a system ease our ability to develop new chunks, by avoiding repetition and ensuring the new chunk is uniquely suited to its purpose, but it also aids in our understanding of which chunks are suited to capture what types of knowledge.

We have developed a classification system for chunks grouping them based on the properties they encode. For example, all chunks encoding things that are measured with units would be instances of the `Unitary` type. Chunks that capture quantities, whether Unitary or not, would be instances of the `Quantity` type. As we simplify, we end up determining what qualifies something as a chunk in the first place. That is, what is the root property that *all* chunks must have to be considered a chunk? That is the `HasUID` classification, which essentially states that a given chunk of knowledge can be referred to by some unique identifier. For a more intuitive example, the `NamedIdea` classification is used for chunks that encode knowledge represented by given terms such as "force", "computer", "priority", and so on.

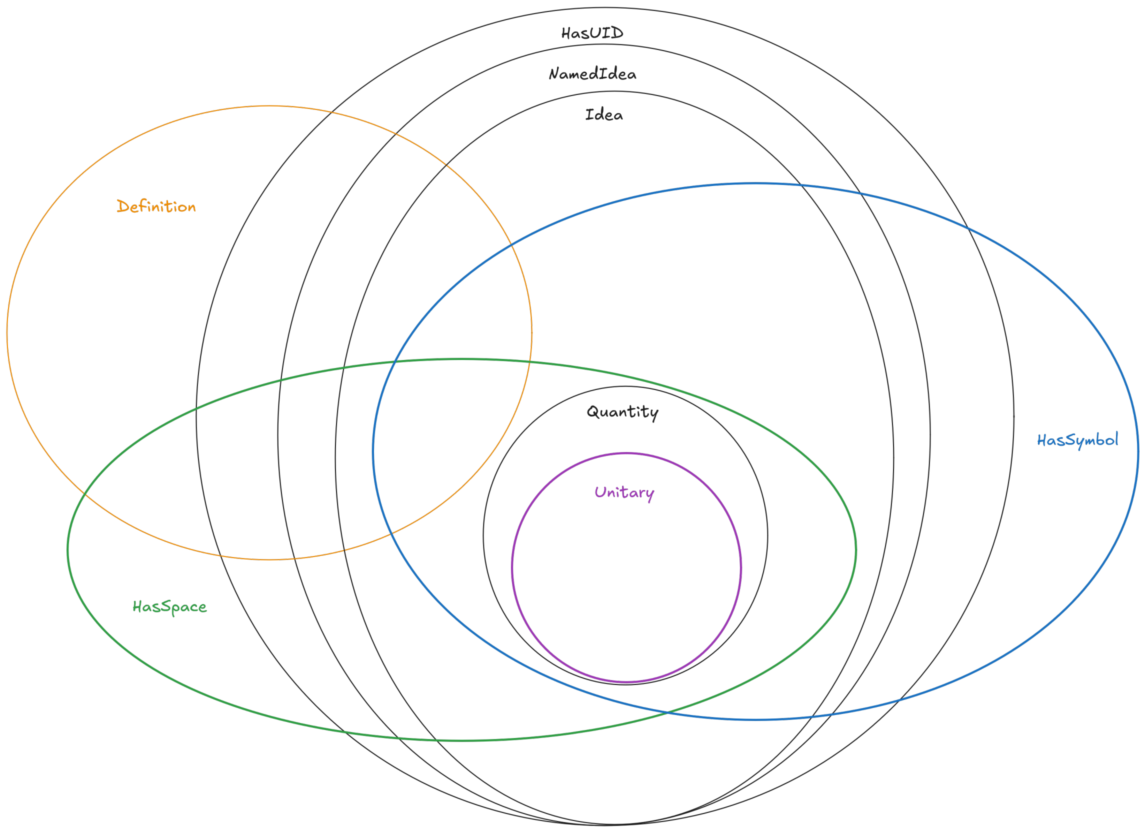


Figure 4.11: A subset of Drasil's Chunk Classification system showing some of the encoded property relationships

Figure 4.11 and Figure 4.12 show a subset of the different ways we classify chunks and some example chunk types and how they would be classified, respectively. Note there is a one to many relationship between a given chunk and its classifications. As you can see, the `UnitalChunk` and `DefinedQuantityDict` are very closely related in that they both contain their own `NamedIdea`, `Space`, `Symbol`, and `Quantity`, however, the `UnitalChunk` is also classified as `Unitary` as it *must* have a unit associated with it.

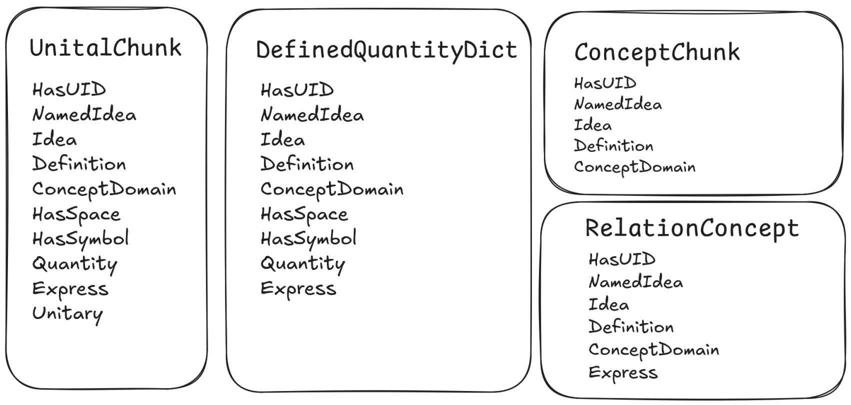


Figure 4.12: A subset of Drasil's chunks showing how different chunks may belong to multiple classifications

We now have a compact, well-typed vocabulary for representing domain knowledge: chunks and classifications give us named concepts, quantities, units, and relationships that are easy to reason about. That vocabulary is only useful if we can turn it into artifacts people consume. To convert that vocabulary into artifacts people actually use, we need a way to operationalize those chunks. In Drasil that means using recipes.

4.3 Recipes: Codifying Structure

Given our chunk classes and a compositional Expr/Sentence model in place, we have a knowledge core that can be selectively projected for any audience. The next step is thus operational: we must describe how those chunks are composed into artifact-specific recipes and how the recipes are rendered into concrete softifacts (documents, code, and supporting files).

Our knowledge base is essentially just a collection of chunks, which are themselves a collection of definitions, encoded in our Expr and Sentence DSLs. To generate softifacts we need a means to define the overall structure of the softifact as well as the knowledge projections we require to expose the appropriate knowledge from our chunks in a meaningful way for the target audience of that particular softifact. This is where our *Recipe* DSL comes into play.

A recipe is, in it's simplest form, a specification for one or more softifacts. It is an intermediate representation of a given softifact and can be thought of as a "little program". With it, we select which knowledge is relevant and how it should be organized before passing the recipe to the generator/rendering engine to be consumed and produce the final softifact. A recipe not only allows us to structure where knowledge should be presented, but also gives us tools for automatically generating non-trivial sections of our softifacts. For example, a traceability matrix can be automatically generated by traversing the tree of relationships between chunks within a recipe, although we'll discuss that in more depth in Section 4.4.

As we saw in Section 3.3, there are patterns of repetition for the way knowledge is projected and organized within and between our softifacts. The recipe DSL has been designed to take advantage of our knowledge capture mechanisms to reduce unnecessary repetition and duplication. We consider the softifacts as different views of the same knowledge, and project only that which is relevant to the audience of the specific softifact our recipe applies to. For example, a human readable document may use symbolic mathematics notation for representing formulae whereas the source code would represent the same formulae using syntax specific to the programming language in use (or, taking an extreme example in the case of punch cards, punched cards). Regardless, we would define the underlying knowledge once in our chunks,

```
-- | A Software Requirements Specification Declaration is made up of all necessary
     sections ('DocSection's).
type SRSDecl = [DocSection]

-- | Contains all the different sections needed for a full SRS ('SRSDecl').
data DocSection = TableOfContents          -- ^ Table of Contents
  | RefSec DL.RefSec                       -- ^ Reference
  | IntroSec DL.IntroSec                   -- ^ Introduction
  | StkhldrSec DL.StkhldrSec               -- ^ Stakeholders
  | GSDSec DL.GSDSec                       -- ^ General System  Description
  | SSDSec SSDSec                          -- ^ Specific System Description
  | ReqrmntSec ReqrmntSec                  -- ^ Requirements
  | LCsSec                                 -- ^ Likely Changes
  | UCsSec                                 -- ^ Unlikely Changes
  | TraceabilitySec DL.TraceabilitySec     -- ^ Traceability
  | AuxConstntSec DL.AuxConstntSec         -- ^ Auxiliary Constants
  | Bibliography                           -- ^ Bibliography
  | AppndxSec DL.AppndxSec                 -- ^ Appendix
  | OffShelfSolnsSec DL.OffShelfSolnsSec -- ^ Off the Shelf Solutions
```

Figure 4.13: Recipe Language for SRS

and use the recipe DSL to select the representation.

[A little repetitive above, but wanted to really hammer in what recipes are for. —DS]

As our case studies are following templates for our softifacts, they give us a solid foundation with which to write our recipes. We already know how the knowledge should be organized, greatly simplifying how we can begin structuring our recipe DSL. The most straightforward, practical approach would be to start by defining a recipe for a given softifact type (SRS, MG, Code, etc.). Then flesh that out section by section, following the template as an organizational guide, using the patterns we discovered earlier (Section 3.3) as a means of avoiding unnecessary duplication. This is the approach we have chosen to follow, and it has lead to some interesting results.

As an example, let us take a look at the recipe language we have defined for an SRS using the Smith et al. template. Looking at the `DocSection` definition from Figure 4.13, we can see a striking resemblance to (a subset of) the template's Table of Contents (Figure 3.1). Each section is defined by a datatype, which is, through a

```haskell
-- | Requirements section (wraps 'ReqsSub' subsections).
newtype ReqrmntSec = ReqsProg [ReqsSub]

-- | Requirements subsections.
data ReqsSub where
  -- | Functional requirements. 'LabelledContent' for tables (includes input values).
  FReqsSub    :: Sentence -> [LabelledContent] -> ReqsSub
  -- | Functional requirements. 'LabelledContent' for tables (no input values).
  FReqsSub'   :: [LabelledContent] -> ReqsSub
  -- | Non-Functional requirements.
  NonFReqsSub :: ReqsSub
```

Figure 4.14: `ReqrmntSec` Decomposition and Definitions

similar mechanism, decomposed into multiple subsections (for example, Figure 4.14 shows the decomposition of the requirements section) which then defines the types of organizational structures we use for specifying the contents of a given subsection. When populated, the recipe specifies both the structure of our expected softifact and the knowledge (chunks) contained therein.

As the given recipe so far has provided nothing but structure, you may be wondering how we populate it with the appropriate knowledge chunks to produce a softifact? For that, we should look at a specific example from one of our case studies (GlassBR).

4.3.1 Example: SRS recipe for GlassBR

The recipe for the GlassBR SRS can be seen in Figure 4.15. This should look familiar, as it follows a similar structure to the Smith et al. template table of contents. We use a number of helper functions and data types to condense and collect the chunks we require and avoid writing any generic boilerplate text. The structure of the document can be rearranged to an extent, in that we can add, remove, or re-order sections at will by simply moving them around within the `SRSDecl`. We can also make choices regarding the display of certain types of information in the final,

```
srs :: Document
srs = mkDoc mkSRS  (S.forGen titleize phrase) si
```

⋮

```
mkSRS = [TableOfContents,
  RefSec $ RefProg intro [TUnits, tsymb [TSPurpose, SymbOrder], TAandA],
  IntroSec $
    IntroProg (startIntro software blstRskInvWGlassSlab glassBR)
      (short glassBR)
    [IPurpose $ purpDoc glassBR Verbose,
     IScope scope,
     IChar [] (undIR ++ appStanddIR) [],
     IOrgSec orgOfDocIntro Doc.dataDefn (SRS.inModel [] []) orgOfDocIntroEnd],
  StkhldrSec $
    StkhldrProg
      [Client glassBR $ phraseNP (a_ company)
        +:+. S "named Entuitive" +:+ S "It is developed by Dr." +:+ S (name
    mCampidelli),
      Cstmr glassBR],
  GSDSec $ GSDProg [SysCntxt [sysCtxIntro, LlC sysCtxFig, sysCtxDesc, sysCtxList],
    UsrChars [userCharacteristicsIntro], SystCons [] [] ],
  SSDSec $
    SSDProg
      [SSDProblem $ PDProg prob [termsAndDesc]
        [ PhySysDesc glassBR physSystParts physSystFig []
        , Goals goalInputs],
       SSDSolChSpec $ SCSProg
        [ Assumptions
        , TMs [] (Label : stdFields)
        , GDs [] [] HideDerivation -- No Gen Defs for GlassBR
        , DDs [] ([Label, Symbol, Units] ++ stdFields) ShowDerivation
        , IMs [instModIntro] ([Label, Input, Output, InConstraints, OutConstraints]
    ++ stdFields) HideDerivation
        , Constraints auxSpecSent inputDataConstraints
        , CorrSolnPpties [probBr, stressDistFac] []
        ]
      ],
  ReqrmntSec $ ReqsProg [
    FReqsSub inReqDesc funcReqsTables,
    NonFReqsSub
  ],
  LCsSec,
  UCsSec,
  TraceabilitySec $ TraceabilityProg $ traceMatStandard si,
  AuxConstntSec $ AuxConsProg glassBR auxiliaryConstants,
  Bibliography,
  AppndxSec $ AppndxProg [appdxIntro, LlC demandVsSDFig, LlC dimlessloadVsARFig]]
```

Figure 4.15: SRS Recipe for GlassBR

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

Figure 4.16: System Information for GlassBR

generated softifact. For example, we alluded to vector representations earlier (Section 4.2.2) which we currently can display using either bold typeface or italics. This choice must be made using the `TypogConvention` datatype which lists the choices made (ex. `TypogConvention[Vector Bold]`) and is used by a relevant portion of the SRS recipe if vectors are included in the softifact. Our GlassBR example does not make use of this convention, however the GamePhysics case study does.

As for the chunks necessary for generating the softifact, each section includes references to them through supplied lists in something we call the `System Information` object (Figure 4.16).

The system information object uses helper functions to keep track of the lists of chunks necessary for filling in the sections of our softifact, the SRS. Each piece of the object consists of one or more chunks to be used in filling in the relevant sections. A quick description of each property can be found in Table 4.1, but it should be noted that some of the defined properties are only there for convenience (and/or remain only until the next refinement pass) as they are derived from other chunks with the

Table 4.1: System information object breakdown - every property is represented by chunks encoding given information

Property	**Chunk(s) Referenced**
_sys	System name (ex. "GlassBR").
_kind	Softifact type we are defining (ex. SRS).
_authors	List of authors of the softifact.
_purpose	Purpose of the softifact.
_quants	List of required quantities.
_concepts	List of required concepts not otherwise encoded.
_instModels	List of required instance models.
_datadefs	List of required data definitions.
_configFiles	List of required configuration files.
_inputs	List of required inputs to the system.
_outputs	List of required outputs from the system.
_constraints	List of constraints for the system (physical and software specific).
_constants	List of required constants.
_sysinfodb	Database of all required chunks.
_usedinfodb	Database of all used acronyms and symbols.
refdb	Database of all relevant external references/citations.

use of helper functions and could be inferred.

[TODO: I might remove the table and just add comments to the code block similar to what's in the next section. It seems like that might be easier for readers to reference —DS]

As should be obvious, the system information object is referencing other chunks which have been defined elsewhere. Most definitions can be found in the knowledge-base, with system-specific aggregations (i.e. lists of chunks) being defined within the scope of the specific example project. For example, `iMods` referenced as the `_instModels` property is defined as shown in Figure 4.17, alongside the two system-specific instance models listed within it. The two specific instance models can be seen to be derived from other existing chunks, through the use of a series of helper

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

⋮

```
pbIsSafe :: InstanceModel
pbIsSafe = imNoDeriv (equationalModelN (nounPhraseSP "Safety Req-Pb") pbIsSafeQD)
  [qwC probBr $ UpFrom (Exc, exactDbl 0), qwC pbTol $ UpFrom (Exc, exactDbl 0)]
  (qw isSafePb) []
  [dRef astm2009] "isSafePb"
  [pbIsSafeDesc, probBRRef, pbTolUsr]
```

⋮

```
lrIsSafe :: InstanceModel
lrIsSafe = imNoDeriv (equationalModelN (nounPhraseSP "Safety Req-LR") lrIsSafeQD)
  [qwC lRe $ UpFrom (Exc, exactDbl 0), qwC demand $ UpFrom (Exc, exactDbl 0)]
  (qw isSafeLR) []
  [dRef astm2009] "isSafeLR"
  [lrIsSafeDesc, capRef, qRef]
```

Figure 4.17: The `iMods` definition

functions, thus ensuring their traceability to the knowledge-base.

Given that the system information object refers to all of the requisite chunks for the knowledge contained in our softifacts and the SRS recipe defines how that knowledge should be structured and organized within the softifact, it follows that those are all we should need to generate our SRS. As an aside, the complete recipe for the SRS specification of GlassBR[4] including the system information object is contained in a single file in less than 370 lines of code[5]. Teasing our results: the generated SRS in PDF format is 44 pages in length. This is partially due to our ability to strip away all of the common boilerplate text that we need not worry about right now, as that will be a problem for the generator (Section 4.4). As it stands we have created the recipe language to represent what we truly want out of a recipe: a simplified, unnecessary-duplication-avoiding, fully traceable representation of our

[4]Not including chunks referenced from within our knowledge bases as they can be shared, reused, and are not necessarily specific to only this system.

[5]Including many comments, whitespace, and import statements.

target softifact.

With the combination of system information and the SRS recipe we can soon move on to creating our generator for the finished SRS document. As the other softifact recipes were developed in tandem with the SRS, they follow a similar structure and in the interest of brevity we will skip the breakdown of each recipe. It is left to the reader to investigate them by diving into our examples via the Drasil github. However, there is one softifact that is significantly different, and as such, we will go into more depth in discussing in the following section: the executable code recipe. Said recipe allows us to not only specify knowledge and the way we wish it structured, but also implementation choices that we would like to make in the final generated source code.

4.3.2 Example: Executable Code recipe for GlassBR

As discussed in the previous section, the executable code recipe is fundamentally different from the documentation-heavy SRS recipe. While the SRS recipe focuses on organizing and projecting knowledge for human consumption, the executable code recipe must specify the structure and content required for generating working source code. This includes not only the relevant knowledge chunks, but also the implementation choices (modules, functions, and other details) necessary for code generation.

The executable-code recipe for GlassBR somewhat mirrors the document recipe in structure but emphasizes implementation choices: module layout, data representation, target languages, optional features, and constraint-handling strategies. Practically, we supply the modules and computational relationships (the mathematical execution order), target languages (e.g. Python, C++, Java), modularity (single file

Body.hs

```haskell
fullSI :: SystemInformation
fullSI = fillcdbSRS mkSRS si
```

Choices.hs

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

choices :: Choices
choices = defaultChoices {
  lang = [Python, Cpp, CSharp, Java, Swift],
  architecture = makeArchit (Modular Separated) Program,
  dataInfo = makeData Bundled Inline Const,
  optFeats = makeOptFeats
    (makeDocConfig [CommentFunc, CommentClass, CommentMod] Quiet Hide)
    (makeLogConfig [LogVar, LogFunc] "log.txt")
    [SampleInput "../../datafiles/glassbr/sampleInput.txt", ReadME],
  srsConstraints = makeConstraints Exception Exception
}
```

Figure 4.18: The Recipe for GlassBR's Executable Code

or separated modules), data bundling (structs/classes), and logging/documentation options.

The recipe for GlassBR's executable code constructed using the `CodeSpec` DSL, is shown in Figure 4.18. Here, the `codeSpec` function takes three arguments:

- `fullSI`: the system information object, which aggregates all the knowledge chunks relevant to GlassBR (as described in the SRS example).

- `choices`: a record specifying code generation options, such as target languages, architecture, data handling, and auxiliary files. The example `choices` definition for GlassBR can be seen in Figure 4.18 and the object itself will be explained in more detail later in this section.

- `allMods`: a list of modules to be generated, each defined in terms of the knowledge chunks and functions they encapsulate.

```haskell
-- | Code specifications. Holds information needed to generate code.
data CodeSpec where
  CodeSpec :: (HasName a) => {
  -- | Program name.
  pName :: Name,
  -- | Authors.
  authors :: [a],
  -- | All inputs.
  inputs :: [Input],
  -- | Explicit inputs (values to be supplied by a file).
  extInputs :: [Input],
  -- | Derived inputs (each calculated from explicit inputs in a single step).
  derivedInputs :: [Derived],
  -- | All outputs.
  outputs :: [Output],
  -- | List of files that must be in same directory for running the executable.
  configFiles :: [FilePath],
  -- | Mathematical definitions, ordered so that they form a path from inputs to
  -- outputs.
  execOrder :: [Def],
  -- | Map from 'UID's to constraints for all constrained chunks used in the problem.
  cMap :: ConstraintCEMap,
  -- | List of all constants used in the problem.
  constants :: [Const],
  -- | Map containing all constants used in the problem.
  constMap :: ConstantMap,
  -- | Additional modules required in the generated code, which Drasil cannot yet
  -- automatically define.
  mods :: [Mod],  -- medium hack
  -- | The database of all chunks used in the problem.
  sysinfodb :: ChunkDB
  } -> CodeSpec
```

Figure 4.19: The Recipe Language for Executable Code (CodeSpec)

Table 4.2: CodeSpec object breakdown

Property	**Chunk(s) Referenced**
pName	Program name
authors	List of authors
inputs	All input variables
outputs	All output variables
configFiles	Required configuration files
execOrder	Mathematical definitions ordered such that they form a path from inputs to outputs
cMap	Constraints on variables
constants	Constants used in the program
mods	List of additional code modules to generate
sysinfodb	Database of all knowledge chunks for the given problem

The `CodeSpec` DSL itself is defined in Figure 4.19, and a subsection of its key fields are summarized in Table 4.2. As mentioned above, we use a helper function `codeSpec` to extract the appropriate fields from the system information, choices, and module list retaining full traceability and avoiding unnecessary manual duplication.

As with the SRS recipe, the code recipe references knowledge chunks defined elsewhere. For example, the list of input variables is defined in `Unitals.hs` just as they were for the SRS. Similarly, the modules to be generated are defined in `ModuleDefs.hs` and make reference to other chunks (both system specific and generic) from our knowledge-base. Looking into the modules, we can see the `readTableMod` module, for example, is defined as shown in Figure 4.20. This module provides a function for reading ASTM glass data from a file, encapsulating both the knowledge of the data format and the implementation logic required for code generation.

While the encapsulation of knowledge in chunks is interesting, we have seen it before in the SRS recipe example. What is novel to the code recipe is the `Choices` object. As mentioned above, it contains the outcome of very important implementation-level

```haskell
readTableMod :: Mod
readTableMod = packmod "ReadTable"
  "Provides a function for reading glass ASTM data" [] [readTable]

readTable :: Func
readTable = funcData "read_table"
  "Reads glass ASTM data from a file with the given file name"
  [ singleLine (repeated [quantvar zVector]) ',',
    multiLine (repeated (map quantvar [xMatrix, yMatrix])) ','
  ]
```

Figure 4.20: The `readTableMod` module definition

```haskell
data Choices = Choices {
  -- | Target languages.
  -- Choosing multiple means program will be generated in multiple languages.
  lang :: [Lang],
  -- | Architecture of the program, include modularity and implementation type
  architecture :: Architecture,
  -- | Data structure and represent
  dataInfo :: DataInfo,
  -- | Maps for 'Drasil concepts' to 'code concepts' or 'Space' to a 'CodeType
  maps :: Maps,
  -- | Setting for Softifacts that can be added to the program or left it out
  optFeats :: OptionalFeatures,
  -- | Constraint violation behaviour. Exception or Warning.
  srsConstraints :: Constraints,
  -- | List of external libraries what to utilize
  extLibs :: [ExtLib]
}
```

Figure 4.21: The `Choices` object definition

choices that must be made to generate the executable code. The definition for the `Choices` object can be seen in Figure 4.21 and a breakdown of each property can be found in Table 4.3.

With our understanding of the `Choices` object, we can now look back at the GlassBR example (Figure 4.18) and understand what specific implementation choices have been made. First off, the `lang` property has been set so the generated code will be output in five different programming languages: Python, C++, C#, Java, and Swift. This is a good demonstration of Drasil's ability to target multiple platforms from a single knowledge base. The `architecture` is configured as modular, with

Table 4.3: Choices object breakdown

Property	
lang	List of target languages
architecture	Architecture of the generated code. Includes modularity (whether to split into modules or generate a single flat file) and implementation type (library to be consumed or standalone program)
dataInfo	Data structure and representation choices. (Ex. bundle inputs together into a class/struct, define constants inline, use the languages constant mechanism, etc.)
maps	Mapping of Drasil concepts and mathematical spaces to code concepts and types in the target language(s) (ex. One could map the concept of π with the language's built-in π and the $\mathbb{R}$ space to single/double precision floating point numbers)
optFeats	Choices for optional features including documentation (comments and verbosity), logging, and auxiliary files.
srsConstraints	Constraint violation behaviour. Can be used to specify whether to throw a warning or exception for physical/software constraints independently.
extLibs	List of external libraries to use (ex. for solving specific classes of mathematics problems). These libraries are external to Drasil and give the option for users to link to optimized, well-established libraries rather than reimplementing them from scratch in Drasil.

input-related functions separated into their own modules (`Modular Separated`), and the implementation type is set to `Program`, meaning the generated code will be a complete, runnable application rather than just a library.

For data representation, the `dataInfo` field specifies that inputs should be bundled together (for example, as a class or struct), constants should be defined inline within the code, and the language's constant mechanism should be used where possible. The `optFeats` field enables comprehensive documentation by including function, class, and module-level comments, but sets the documentation verbosity to quiet and hides date fields in the generated comments. Logging is also enabled for both variable assignments and function calls, with all logs directed to a file named `log.txt`. Additionally, the auxiliary files generated include a sample input file (pointing to a real data file) and a `README`, supporting both usability and reproducibility.

Finally, the `srsConstraints` field is set so that any violation of software or physical constraints will result in an exception, ensuring that errors are caught and handled strictly during execution. This configuration ensures that the generated GlassBR code is well-documented, maintainable, and ready for use in a variety of environments.

By structuring the executable code recipe in this way, Drasil ensures that all generated code is fully traceable to the underlying knowledge base and any external libraries. Any change in the knowledge chunks or their relationships is automatically reflected in the generated code, just as it is in the documentation. This approach minimizes duplication, maximizes consistency, and enables reliable code generation for scientific computing applications.

4.3.3 Operational Summary

Having established how Drasil captures, organizes, and operationalizes knowledge through both documentation and executable code recipes, we are now poised to explore the final step in Drasil's operation: turning these structured specifications into tangible softifacts. The next piece of the story will show how Drasil takes these recipes and, through a unified process, produces the diverse softifacts required by different stakeholders. In the following section, we delve into the mechanisms and philosophy behind Drasil's generation and rendering, showing how the framework brings together all the ingredients we've discussed to deliver consistent, traceable, and maintainable softifacts.

4.4 Cooking it all up: Generation/Rendering

The previous sections have detailed how Drasil captures domain knowledge as chunks, organizes it via a robust hierarchy, and structures it into recipes that specify the desired artifacts. In this section, we focus on the final stages of the generation pipeline: *rendering* and *assembly*. These stages are responsible for transforming the intermediate representations (recipes and their referenced chunks) into tangible, concrete, consumable softifacts, such as documents and executable code.

4.4.1 Rendering: From Abstract Structure to Concrete Formats

At its core, Drasil treats recipes as "little programs". Rendering in Drasil refers to the process by which these abstract, language-agnostic structures defined in recipes are transformed into specific output formats suitable for end-users and stakeholders. This stage bridges the gap between the high-level, reusable knowledge representations and the diverse forms required for practical use (ex. LaTeX, HTML, PDF, or source code in various programming languages).Drasil achieves this through a set of dedicated *printers* and, for code, through the use of the General Object-Oriented Language (GOOL)[?] library. Each printer is designed to interpret the structured data produced by the recipe and knowledge projection stages and emit output in the desired format, handling both content and layout[6].

Document Rendering: Printers and Output Engines

For documentation artifacts (such as SRS or Module Guide), Drasil provides specialized printers for each supported format:

- `genTeX` in `Language.Drasil.TeX.Print` — renders the document as LaTeX source, suitable for PDF generation.
- `genHTML` in `Language.Drasil.HTML.Print` — produces HTML for web-based consumption.
- `genJSON` in `Language.Drasil.JSON.Print` — outputs a machine-readable JSON representation, supporting further processing or integration.

Each printer traverses the recipe structure, visiting each section and sub-section, and calls formatting routines appropriate to the target format. For example, a Data Definition chunk will be rendered as a LaTeX table row, an HTML table entry, or a JSON object, depending on the printer invoked.

[6]To some respect, depending on the desired output format. For instance, LaTeXhandles the majority of the finished softifact's layout where Drasil generates the content

```haskell
-- | Generates a LaTeX document.
genTeX :: L.Document -> PrintingInformation -> TP.Doc
genTeX doc@(L.Document _ _ toC _) sm =
  runPrint (buildStd sm toC $ I.makeDocument sm $ L.checkToC doc) Text
```

Figure 4.22: `genTeX` function for LaTeX rendering

Example: SRS to LaTeX

As a concrete illustration, consider the printing of the GlassBR SRS to LaTeX. The process begins with the invocation of the `genTeX` function, which recursively walks the document tree produced by the recipe, emitting LaTeX code for sections, equations, and tables. Figure 4.22 shows the entry point to this process.

Code Rendering: GOOL and Target Language Emission

For executable code artifacts, Drasil uses GOOL[?] as an intermediate layer. GOOL provides a collection of type-safe combinators and abstractions for representing object-oriented and procedural constructs in a language-agnostic manner. The code recipe (ex. a `CodeSpec`) is first translated into a GOOL abstract syntax tree (AST), which is then emitted as source code in one or more target languages. For brevity we will not get into the implementation details regarding Drasil's use of GOOL, as those have been covered in other works [?] [Someone else's thesis covered this yeah? I don't want to make it sound like I wrote this part, but still want to give an overview —DS].

In brief, the code emission process follows a straightforward translation process: The code recipe and its referenced chunks are traversed to construct a GOOL AST, representing classes, functions, variables, and control flow; The GOOL backend is invoked for each specified target language, translating the AST into concrete source code (ex. Python, C++, Java, C#); Auxiliary files (such as README, Doxygen

```haskell
-- | Generates a package with the given 'DrasilState'. The passed
-- un-representation functions determine which target language the package will
-- be generated in.
generateCode :: (OOProg progRepr, PackageSym packRepr) => Lang ->
  (progRepr (Program progRepr) -> ProgData) -> (packRepr (Package packRepr) ->
  PackData) -> DrasilState -> IO ()
generateCode l unReprProg unReprPack g = do
  workingDir <- getCurrentDirectory
  createDirectoryIfMissing False (getDir l)
  setCurrentDirectory (getDir l)
  let (pckg, ds) = runState (genPackage unReprProg) g
      code = makeCode (progMods $ packProg $ unReprPack pckg)
        ([ad "designLog.txt" (ds ^. designLog) | not $ isEmpty $
          ds ^. designLog] ++ packAux (unReprPack pckg))
  createCodeFiles code
  setCurrentDirectory workingDir
```

Figure 4.23: generateCode and genPackage for code rendering

config, and sample input files) are generated via dedicated routines.

Example: Emitting GlassBR Code in Python

For GlassBR, suppose the code recipe specifies Python as a target. The generation process calls `generateCode` in `Language.Drasil.Code.Imperative.Generator`, which constructs the GOOL AST and then emits Python code by invoking the Python backend. The resulting code includes all functions, data structures, and comments specified in the recipe and choices object, as well as any automatically generated traceability links (ex. comments referencing requirements or data definitions). Figure 4.23 shows the relevant code path.

4.4.2 Assembly: Producing the Final Softifact Set

Once rendering is complete, the generated document or code sections must be assembled into coherent artifacts ready for distribution and use. Assembly includes writing the primary artifacts to disk (ex. *.tex, *.html, *.py, *.cpp files), generating auxiliary files (ex. Makefiles, CSS, README, example input files), linking cross-references,

```haskell
-- document information and printing information. Then generates the document file.
prntDoc' :: String -> String -> Format -> Document -> PrintingInformation -> IO ()
prntDoc' dt' fn format body' sm = do
  createDirectoryIfMissing True dt'
  outh <- openFile (dt' ++ "/" ++ fn ++ getExt format) WriteMode
```

Figure 4.24: `prntDoc'` function for document output

```haskell
-- | Helper for writing the Makefile(s).
prntMake :: DocSpec -> IO ()
prntMake ds@(DocSpec (DC dt _) _) =
  do outh <- openFile (show dt ++ "/PDF/Makefile") WriteMode
     hPutStrLn outh $ render $ genMake [ds]
     hClose outh
```

Figure 4.25: `prntMake` function for document output

and ensuring the integrity of tables, figures, and code modules. The remainder of this section illustrates these steps in practice.

Example: SRS Assembly

After rendering the GlassBR SRS to LaTeX via the `genTeX` function seen in Figure 4.22, the `prntDoc'` function, Figure 4.24, writes the source file, while `prntMake`, Figure 4.25, generates a Makefile for automated PDF compilation. The resulting directory contains a fully cross-referenced, typeset-ready SRS and all supporting files.

Example: Code Assembly

Once the code has been rendered through the GOOL pipeline into target-language text, the next stage is to assemble the resulting files into a coherent project structure. In this case, `createCodeFiles` (Figure 4.26) is called to write each generated source file and all auxiliary documentation to disk using consistent directory and naming conventions. For multi-language builds, this process is repeated for each target language, ensuring consistency across all output.

```haskell
-- | Creates the requested 'Code' by producing files.
createCodeFiles :: Code -> IO () -- [(FilePath, Doc)] -> IO ()
createCodeFiles (Code cs) = mapM_ createCodeFile cs

-- | Helper that uses pairs of 'Code' to create a file written with the given
    document at the given 'FilePath'.
createCodeFile :: (FilePath, Doc) -> IO ()
createCodeFile (path, code) = do
  createDirectoryIfMissing True (takeDirectory path)
  h <- openFile path WriteMode
  hPutStrLn h (render code)
  hClose h
```

Figure 4.26: createCodeFiles for code output

In essence, this stage performs for code what the SRS assembly stage does for documentation: it gathers the individually generated fragments and organizes them into a complete, ready-to-use product. The result comprises all files necessary for the generated system source code, build instructions, and supporting assets. They are generated and ready for compilation or integration into a development environment. Because these outputs are directly derived from the same knowledge base that informed the SRS, Drasil ensures that the implementation remains consistent with the documentation.

4.4.3 A GlassBR Example: From Chunk Definition to Final SRS Rendering

To illustrate the rendering and assembly process, we follow a single chunk through each step of generation in the GlassBR SRS. Earlier, we described how the SRS recipe for GlassBR is constructed by specifying the document structure and referencing the necessary knowledge chunks through the system information object (see Section 4.3.1). Now we observe the journey of the dimensionless load chunk ($\hat{q}$), from its definition in the knowledge base to its appearance as typeset-ready LaTeXformatted

```
--DD7--

dimLLEq :: Expr
dimLLEq = mulRe (sy demand) (square (mulRe (sy plateLen) (sy plateWidth)))
  $/ mulRe (mulRe (sy modElas) (sy minThick $^ exactDbl 4)) (sy gTF)

dimLLQD :: SimpleQDef
dimLLQD = mkQuantDef dimlessLoad dimLLEq

dimLL :: DataDefinition
dimLL = ddE dimLLQD [dRef astm2009, dRefInfo campidelli $ Equation [7]] Nothing "
    dimlessLoad"
  [qRef, aGrtrThanB, stdVals [modElas], hRef, gtfRef]
```

Figure 4.27: Definition of $\hat{q}$ as a chunk in the GlassBR knowledge base

text in the generated SRS.

Step 1: Knowledge Capture

The definition of $\hat{q}$ is captured in the Drasil knowledge base as a chunk (Figure 4.27). This chunk contains all necessary information: a unique identifier, symbol, natural language description, units, and the defining mathematical expression.

Step 2: Recipe Specification

In this step, the previously defined $\hat{q}$ chunk is referenced and included in the system information and recipe for the GlassBR SRS. Here, $\hat{q}$ is added to the list of Data Definitions (can be seen as `GB.dataDefs` in Figure 4.16), which will later be used to populate the relevant sections (Table of Symbols, Data Definitions, Traceability Matrix, etc.) of the document. This step is about specifying what knowledge is relevant and where it should appear, not about how it is rendered.

Step 3: Generation and Rendering

When the generator is invoked to produce the SRS (ex. as a LaTeX document), it traverses the recipe, visiting each section. For the Data Definitions section, it projects the full definition of $\hat{q}$, including its symbol, description, units, and defining equation. For the Table of Symbols, it projects only the symbol, a brief description, and the units. The rendering engine formats these projections according to the conventions of the target output.

During rendering, the SRS generator traverses the recipe and, for each chunk referenced, projects and formats it for the target output. For the Data Definitions section, it projects the full definition of $\hat{q}$, including its symbol, description, units, and defining equation (Figure 4.28). For the Table of Symbols, it projects only the symbol, a brief description, and the units. The rendering engine formats these projections according to the conventions of the target output (as a LaTeXtable or section).

Step 4: Assembly and Output

Finally, the rendered sections are assembled into the complete SRS artifact. Cross-references are automatically generated, so that, for example, references to $\hat{q}$ in Instance Models or Requirements link back to its definition in the Data Definitions and/or Table of Symbols. Auxiliary materials, such as the Table of Units and Table of Abbreviations, if required, are generated by traversing the relevant lists in the system information object, ensuring consistency and completeness. The completed document is output as a `.tex` file with $\hat{q}$ fully integrated and accessible alongside all other relevant auxiliary files (ex. `.bib` file for citations). A makefile is also generated

```
\vspace{\baselineskip}
\noindent
\begin{minipage}{\textwidth}
\begin{tabular}{>{\raggedright}p{0.13\textwidth}>{\raggedright\arraybackslash}p{0.82\
    textwidth}}
\toprule \textbf{Refname} & \textbf{DD:dimlessLoad}
\phantomsection
\label{DD:dimlessLoad}
\\ \midrule \\
Label & Dimensionless load

\\ \midrule \\
Symbol & $\hat{q}$

\\ \midrule \\
Units & Unitless

\\ \midrule \\
Equation & \begin{displaymath}
           \hat{q}=\frac{q \left(a b\right)^{2}}{E h^{4} \mathit{GTF}}
           \end{displaymath}
\\ \midrule \\
Description & \begin{symbDescription}
              \item{$\hat{q}$ is the dimensionless load (Unitless)}
              \item{$q$ is the applied load (demand) (${\text{Pa}}$)}
              \item{$a$ is the plate length (long dimension) (${\text{m}}$)}
              \item{$b$ is the plate width (short dimension) (${\text{m}}$)}
              \item{$E$ is the modulus of elasticity of glass (${\text{Pa}}$)}
              \item{$h$ is the minimum thickness (${\text{m}}$)}
              \item{$\mathit{GTF}$ is the glass type factor (Unitless)}
              \end{symbDescription}
\\ \midrule \\
Notes & $q$ is the 3 second duration equivalent pressure, as given in \hyperref[DD:
    calofDemand]{DD:calofDemand}.

        $a$ and $b$ are the dimensions of the plate, where ($a\geq{}b$).

        $E$ comes from \hyperref[assumpSV]{A:standardValues}.

        $h$ is defined in \hyperref[DD:minThick]{DD:minThick} and is based on the
    nominal thicknesses.

        $\mathit{GTF}$ is defined in \hyperref[DD:gTF]{DD:gTF}.

\\ \midrule \\
Source & \cite{astm2009} and \cite[(Eq. 7)]{campidelli}

\\ \midrule \\
RefBy & \hyperref[DD:stressDistFac]{DD:stressDistFac}

\\ \bottomrule
\end{tabular}
\end{minipage}
```

Figure 4.28: Raw generated LaTeXof the $\hat{q}$ Data Definition

to enable the user to easily create a human-readable PDF file from the LaTeXsource and auxiliary files.

4.4.4 Multiple Renderings from a Single Source: From Chunk to Code

As a continuation of the previous sections example, we now follow the dimensionless load $\hat{q}$ from its knowledge base definition through to its realization in the generated source code. This example concretely demonstrates how Drasils single-source-of-truth approach ensures that mathematical concepts are faithfully and consistently reflected in both documentation and code artifacts.

Referencing $\hat{q}$ in the Code Recipe

Recall that $\hat{q}$ was previously defined as a chunk containing its symbol, natural language description, units, and mathematical formula (see Figure 4.27). In the process of generating executable code, this chunk is referenced in the system information object (`fullSI`) and included in the `CodeSpec` recipe for GlassBR(Figure 4.18). The recipe dictates which variables and formulas are required in the target program, and provides implementation-level choices such as type mapping and function structure.

Rendering via GOOL

During code generation, Drasil translates the recipe and referenced chunks into a GOOL (General Object-Oriented Language) intermediate representation. The $\hat{q}$ chunk, for example, is mapped to a function definition within the GOOL AST, preserving its formula and metadata. This intermediate form enables Drasil to emit code

```python
## \brief Calculates dimensionless load
# \param inParams structure holding the input values
# \param q applied load (demand): 3 second duration equivalent pressure (Pa)
# \return dimensionless load
def func_q_hat(inParams, q):

    return q * (inParams.a * inParams.b) ** 2.0 / (7.17e10 * inParams.h ** 4.0 *
    inParams.GTF)
```

Figure 4.29: $\hat{q}$ as defined in the generated python code

in multiple target languages without duplicating logic. As GOOL is not the subject of this thesis, we will elide the GOOL-specific details and continue down the code generation pipeline.

Generated Source Code

In the final step, the GOOL backend emits concrete code for $\hat{q}$ in each selected language. For example, in Python, $\hat{q}$ appears as a function computing its value from its dependencies(Figure 4.29) just as we've seen before in Figure 3.6. The generated code maintains traceability through naming conventions and comments drawn from the original chunk. The final python output can be seen in Figure 4.29.

Discussion

This example demonstrates how Drasils architecture ensures that any update to the definition or formula of $\hat{q}$ is automatically propagated to the generated code, just as it is to documentation artifacts. The use of a knowledge-centric pipeline, combined with language-agnostic intermediate representations, guarantees consistency and traceability throughout the software system. Changes need only be made once, and are reflected everywhere $\hat{q}$ appears, reducing maintenance cost and risk of error.

4.4.5 Summary

In summary, rendering and assembly are the final, crucial steps that operationalize Drasil's knowledge-centric approach and translate the abstract representations produced by our chunks and recipes into practical, high-quality softifacts. The examples from GlassBR demonstrate the practical effectiveness of this approach for both documentation and code.

The next section will discuss how iteration and refinement were used in the creation of Drasil, and how the framework enables rapid evolution of both knowledge and artifacts.

4.5 Iteration and refinement

With the understanding of Drasil's structure and implementation, we can now discuss continuous improvement. The development of Drasil did not follow anything even remotely close to the waterfall model of software development. If anything its development began with a simple set of ideas that created a basis for refinement. That is to say: it *barely* worked.

We approached the development knowing the scope was far too large to plan out all the details ahead of time (in any reasonable time-frame), and made conscious choices on compromises we were willing to accept to get a minimum viable framework and continuously improve upon it, similar to agile software development practices. As such, and as we have eluded to throughout this and the previous chapter, we took a very practical approach to development by starting with a known-good state (our case studies), generalizing as much as possible (as seen in Chapter 3) to distill what we needed to understand to generate software, and updating the framework as our assumptions were challenged or found wanting. Finding holes is a lot easier when you constantly step in them.

Many of our updates involved modifying implementation details (such as Chunk types, combinators, and adding entirely new concepts) when we started looking at the various case study domains and determining how we'd distill and capture the knowledge necessary to reproduce those case studies.

Throughout this process of iterative refinement, we inevitably found a number of places to improve upon the existing case studies and applied those changes to our known-good state. We found errors and undocumented assumptions in the case studies that caused us to rework and improve them for the future. Crucially, moving the case studies into Drasil made those mistakes trivial to locate and correct. A representative example is the SSP symbol inconsistency (Issue #348 on the GitHub issue tracker). The original SSP SRS used the pair S_i and P_i (mobilized shear stress and resistive shear force respectively) in one place and later switched S_i to τ_i (resistive shear stress). τ_i was also omitted from the Table of Symbols. To reiterate, that inconsistency existed in the original case study and was not a generator bug but an error in the source document we had accepted as our starting point. Once the SSP material was encoded as chunks in Drasil the fix was minimal: ensure we had encoded the correct definitions of S_i, P_i, and τ_i and used them in the appropriate equations. Then the correction automatically propagated to all generated artifacts (Table of Symbols, crossreferences, and any code that referenced the symbol).

We also noticed a unit mismatch during the investigation of that issue, which sparked some confusion until we determined there was an unspecified assumption of 1m depth "into the page" for one of the equations. That assumption was later captured and encoded into the specific knowledge for that family of software systems as well. Overall the effort to fix both of these related issues was essentially a focused edit to a minuscule subset of the captured knowledge, not a widespread rewrite.

This pattern repeats across many issues: we would find a mismatch between a case study and our Drasil output, or some as-yet unencoded knowledge (ex. implicit assumptions) that made the generated artifacts inconsistent or unclear, update the chunk(s) that encoded the offending knowledge, and regenerate. That workflow was the dominant mode of refinement of the original case studies and their Drasil implementations. It kept fixes small, localized, and low cost while producing broad, immediate benefits in every artifact that depended on the corrected knowledge.

As Drasil encodes both semantics (definitions, units, relations) and presentation choices (how a chunk is projected), correcting semantics upstream was also used to eliminate downstream inconsistencies with, often, minimal engineering effort. One example of this is changing our Chunk hierarchy to include derived-unit chunks. After we determined the need for `Unital` chunks which tied units to captured knowledge, it was obvious that the units themselves would need to be able to represent their relationships, particularly for those derived from standard SI units (ex. m/s^2). This involved the addition of new combinators for creating `UnitDefn` chunks (examples of some of these combinators have already been shown in Figure 4.7).

In short, we learned how to improve Drasil by trying to reproduce existing case studies, encoding what we needed as chunks, and then correcting or extending the chunks when mismatches were discovered. Fixes were either small edits to the knowledge base (cheap to apply, and automatically, consistently propagated to all generated outputs) or an extension of the way we chose to capture knowledge (new chunk types or combinators to encode previously un-encodable knowledge or relationships between chunks).

4.6 Summary

In this chapter we have described the architecture and implementation choices that make Drasil practical: a knowledgecentric pipeline built from reusable, typed "chunks", lightweight DSLs for expressions and natural language fragments, and a recipe language that codifies how knowledge is projected into concrete artifacts. The design is intentionally pragmatic: we capture what is needed to reproduce real case studies, provide welltyped building blocks for assembling that knowledge, and offer generators (printers and the GOOL back end) that render the same source into multiple, consistent outputs. The result is a single source of truth that spans documentation and executable code.

By separating the concerns of knowledge capture, structuring, rendering, and assembly, Drasil ensures that softifacts remain maintainable, consistent, traceable, and free of unnecessary manual duplication from their high-level specification down to their concrete instances. All of which increases confidence in the resulting software systems.

The chunk hierarchy and its classification system give us the levers we need to balance expressivity and reuse. Simple chunks (NamedChunk, ConceptChunk) let us represent terminology and definitions; richer chunks (UnitalChunk, UnitDefn, DataDefn, InstanceModel) let us attach units, symbols, and executable expressions; and the Expr / Sentence DSLs give us languageagnostic ways to express mathematics and narrative. Recipes then select and arrange those chunks for particular stakeholders so that the same underlying knowledge can be rendered as an SRS, a module guide, or multilanguage source code with full traceability between pieces.

The iterative approach to Drasil's development ensured modularized components that improve extensibility and maintainability of the system as a whole, as well as continuous refinement and expansion of the system's capabilities. A key practical lesson is that encoding case studies into Drasil exposes errors and implicit assumptions in the original artifacts but makes them trivial to fix. Fixes are usually local edits to the knowledge base and those edits automatically propagate through all generated artifacts. This demonstrates the principal payoff of the singlesource, chunkbased approach: small, cheap changes at the source yield broad, consistent improvements everywhere.

Drasil is not a universal solution it is a tool targeted at domains where the models are well understood, longlived, and shared across multiple stakeholders. Within that scope, Drasil shifts effort from repetitive editing of many artifacts to careful modeling of domain knowledge. The framework we have presented allows that modeling to pay dividends immediately (consistent documents and code) and continuously (streamlined evolution through iteration and refinement).

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 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.

/dsDropping this bit from the last section (pre-edits) here for future refinement into something that makes sense. We used examples in Iteration and Refinement, but they make more sense to be here in the grand scheme

This pattern repeats across many issues: we would find a mismatch in a case study or an implicit assumption that made the generated artifacts inconsistent or unclear, update the chunk(s) that encoded the offending knowledge, and regenerate. Because recipes and printers project the same chunks in different forms, a single, small correction in the knowledge base yields consistent fixes across documentation and code. That workflow detect in the generated output, repair the chunk, regenerate was the dominant mode of refinement. It kept fixes small, localized, and low cost while producing broad, immediate benefits in every artifact that depended on the corrected knowledge.

Looking at the issue tracker helps make this concrete: early reports and discussions often reveal errors or unstated assumptions in the original examples rather than systemic generator defects. Fixes typically involved clarifying identifiers, aligning symbol definitions, or making implicit domain choices explicit in the chunks. Because Drasil encodes both semantics (definitions, units, relations) and presentation choices (how a chunk is projected), correcting the semantics upstream eliminated the downstream inconsistencies with minimal engineering effort.

Some of the refinements we have made are straightforward bug fixes to the captured knowledge; others point to design improvements that we should undertake to reduce future fragility or capture broader scopes of knowledge. For instance, Issue #91 (regarding "Parameterized Unitals") highlights a recurring modeling pain: there is no internal concept of a "physical material" with specific properties like mass, density, etc. In the SWHS example we treated "mass of water" and "mass of PCM" as separate adhoc things in different, when they should be views of a single, parameterizable concept ("mass of a physical material"). That issue is an example of a broader-scoped change to the knowledge-capture mechanics and representation within Drasil itself.

In short, our development process has been practical and incremental. We learned by trying to reproduce existing case studies, encoded what we needed as chunks, and then corrected the chunks when mismatches were discovered. Fixes were usually small edits to the knowledge base, cheap to apply, and thanks to the singlesource approach automatically and consistently propagated to all generated outputs. The issue tracker documents both the immediate, lowcost corrections (like #348) and the longerterm modelling improvements we should pursue (like #91).

5.1.1 Guarantees and Extensibility

The rendering and assembly phases are designed to be fully deterministic and repeatable. Because Drasil relies on Haskell's type system and the structure of recipes and chunks, all generated outputs must be *consistent by construction.* Any modification to a chunk or recipe is automatically reflected in every rendered artifact on the next generation pass, eliminating the risk of out-of-date documentation or code. Moreover, the modular design of printers and the GOOL backend makes it straightforward to add new output formats or languages, further enhancing Drasil's long-term utility and extensibility.

5.1.2 Traceability and Consistency

Drasil achieves traceability not by embedding explicit metadata in its generated artifacts, but by making the entire generation process fully deterministic [This will be important later! —DS] and rooted in a single source of truth. All documents and code are produced by traversing recipes and knowledge chunks, ensuring that every output can be traced (by construction) back to the precise definitions and relationships captured in the knowledge base. Cross-references, such as those in tables of symbols or data definitions, are generated automatically as part of this traversal, ensuring human-readable traceability throughout the artifacts.

Crucially, Drasil leverages Haskell's strong static type system to enforce consistency and prevent errors. The chunk hierarchy, smart constructors, and typeclass constraints ensure that only well-formed, type-correct chunks can be constructed and referenced in recipes. As a result, malformed or missing chunks are almost always detected at compile time, rather than at runtime or during artifact generation. This consistency by construction philosophy means that errors are typically surfaced as type errors or missing imports during development, rather than as failures during the generation phase.

Thus, Drasil does not attempt to recover from or annotate malformed data at runtime; instead, it is architected to make such situations impossible (or at least extremely rare) through the design and use of Haskell's type system. This approach provides strong guarantees that all generated softifacts are consistent, complete, and fully traceable to their source knowledge.

5.2 "Pervasive" Bugs

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

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

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

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

and its code representation (in python):

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

At a glance, are the code and formula equivalent?

It is difficult to say without confirming the value of E is defined as $7.17 \cdot 10^{10}$, 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.

5.3 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).

5.4 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]

5.5 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 Inference

[something like the following —DS] - As mentioned earlier throughout Section 4.5, and explicitly stated in Section 4.3.1, there are areas where Drasil can stand to be refined further, particularly around inferring necessary chunks as opposed to explicitly defining them in lists. Taking the SRS as an example, we should be able to refine the system to the point where the System Information object can be simplified by inferring necessary chunks from the SRS Recipe definition. The current system is used namely due to the rigidity of Haskell's type system. Figure 4.19 - `constMap` is another example we should be able to infer eventually (hence the "yet").

6.2 Typed Expression Language

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

6.3 Model types

6.4 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].

6.5 Variants

[may not need a section¡ I just don't want to forget this —DS]

As we see in English, along with many other languages, there is no one way to define something. Having multiple variants of a natural language description for a chunk would be useful as each variant can be used in a different context for different effect to a given audience.

The Newton's Second Law example from Section 4.2.2 shows a perfect example of this as the "loosely translated" description is not the description we have chosen to encode in our chunk, yet both are equally valid.

6.6 Many more artifacts/more document knowledge

Journal papers, Jupyter notebooks, lesson plans, etc.

6.7 More display variabilities

6.8 More languages (?)

[Programming and natural languages. —DS]

6.9 More scientific knowledge

As Drasil sees more use, the knowledge-base will inevitably expand to encompass more knowledge within fields. We would also like to expand it into other scientific fields such as chemistry, medicine, etc.

6.10 More computational knowledge

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

More knowledge around software engineering and design processes.

6.11 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

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`