List of Figures

List of Tables

Chapter 1

A Rational Document Driven Design Process for Scientific Software

W. Spencer Smith

McMaster University, Computing and Software Department

1.1 Introduction

This chapter motivates, justifies, describes and evaluates a rational document driven design process for scientific software. The documentation is adapted from the waterfall model [17, 60], progressing from requirements, to design, to implementation and testing. Many researchers have stated that a document driven process is not used by, nor suitable for, scientific software. These researchers argue that scientific developers naturally use an agile philosophy [1, 5, 13, 47], or an amethododical process [25], or a knowledge acquisition driven process [26]. Just because a rational process is not currently used does not prove that it is inappropriate, only that past efforts have

been unsuccessful. The problem could be inadequate customization to the needs of the scientific community and incomplete training of practitioners. To date, the "from the textbook" Software Engineering (SE) approach may have failed, but that does not mean it should be abandoned. With some modification to suit scientific needs, the benefits of traditional SE can be realized. For instance, a rational design process can provide quality improvements, and Quality Assurance (QA), as shown in Table 1.1. Moreover, documentation, written before and during development, can provide many benefits [42]: easier reuse of old designs, better communication about requirements, more useful design reviews, easier integration of separately written modules, more effective code inspection, more effective testing, and more efficient corrections and improvements.

One argument against a document driven process is that scientists do not view rigid, process-heavy approaches, favorably [5]. As an example from a scientific software developer, Roache [45, p. 373] considers reports for each stage of software development as counterproductive. However, the reports are only counterproductive if the process used by the scientists has to follow the same waterfall as the documentation. This does not have to be the case. Given the exploratory nature of science, developers do not typically follow a waterfall process, but, as Parnas and Clements [41] point out, the most logical way to present the documentation is still to "fake" a rational design process. "Software manufacturers can define their own internal process as long as they can effectively map their products onto the ones that the much simpler, faked process requires" [34]. Reusability and maintainability are important qualities for scientific software. Documentation that follows a faked rationale design process is easier to maintain and reuse because the documentation is understandable and standardized. Understandability is improved because the faked documentation only includes the "best" version of any artifacts, with no need to incorporate confusing details around the history of their discovery [41]. Standardization on any process, with a rational process being a logical choice as a standard, facilitates design reviews, change management and the transfer (and modification) of ideas and software between projects [41].

Another argument against a rational process, where software is derived from precise requirements, centers around the opinion that, in science, requirements are impossible to determine up-front, because the details can only emerge as the work progresses [5, 48]. Is science really so different from other software domains? No, requirements are challenging for every domain. The differences between science and other domains might actually make producing a first draft of the requirements easier. For instance, scientific requirements, in terms of physical models, have a high potential for reuse [54]. The laws of physics are stable; they are almost universally accepted, well understood, and slow to change. At the appropriate abstraction level, many problems have significant commonality, since a large class of physical models are instances of a relatively small number of conservation equations (conservation of energy, mass and momentum). Moreover, scientific software does not typically have

TABLE 1.1: Improving Scientific Software Qualities via Rational Design

Verifiability involves "solving the equations right" [45, p. 23]; it benefits from rational documentation that systematically shows, with explicit traceability, how the governing equations are transformed into code.
Validatability means "solving the right equations" [45, p. 23]. Validatability is improved by a rational process via clear documentation of the theory and assumptions, along with an explicit statement of the systematic steps required for experimental validation.
Usability can be a problem. Different users, solving the same physical problem, using the same software, can come up with different answers, due to differences in parameter selection [45, p. 370]. To reduce misuse, a rational process must state expected user characteristics, modelling assumptions, definitions and the range of applicability of the code.
Maintainability is necessary in scientific software, since change, through iteration, experimentation and exploration, is inevitable. Models of physical phenomena and numerical techniques necessarily evolve over time [5, 48]. Proper documentation, designed with change in mind, can greatly assist with change management.
Reusability provides support for the quality of reliability, since reliability is improved by reusing trusted components [12]. (Care must still be taken with reusing trusted components, since blind reuse in a new context can lead to errors, as dramatically shown in the Ariane 5 disaster [37, p. 37–38].) The odds of reuse are improved when it is considered right from the start.
Understandability is necessary, since reviewers can only certify something they understand. Scientific software developers have the view "that the science a developer embeds in the code must be apparent to another scientist, even ten years later" [25]. Understandability applies to the documentation and code, while usability refers to the executable software. Documentation that follows a rational process is the easiest to follow.
Reproducibility is a required component of the scientific method [9]. Although QA has, "a bad name among creative scientists and engineers" [45, p. 352], the community need to recognize that participating in QA management also improves reproducibility. Reproducibility, like QA, benefits from a consistent and repeatable computing environment, version control and separating code from configuration/parameters [9].

to deal with concurrency (except for the case of parallel processing), real-time constraints, or complex user interactions. The typical scientific software design pattern is simply: Input ⇒ Calculate ⇒ Output. All domains struggle with up-front requirements, but scientists should remember that their requirements do not have to be fully determined a priori. As mentioned in the previous paragraph, iteration is inevitable and a rational process can be faked.

Although current practice tends to neglect requirements documentation, it does not have to be this way. To start with, when researchers say that requirements emerge through iteration and experimentation, they are only referring to one category of scientific software. As observed previously [26, 52], scientific software can be divided into two categories: specific physical models and general purpose tools. When scientific software is general purpose, like a solver for a system of linear equations, the requirements should be clear from the start. General purpose tools are based on well understood mathematics for the functional requirements, as shown in scientific computing textbooks [19]. Even the nonfunctional requirements, like accuracy, can be quantified and described through error analysis and in some cases validated computing, such as interval arithmetic.

Even specialized software, like weather prediction or structural analysis, can be documented a priori, as long as the author's viewpoint takes into account separation of concerns, a broad program family approach and consideration for future change management. With respect to separation of concerns, the physical models should be clearly separated from the numerical methods. Knowing the most appropriate numerical technique is difficult at the outset, but this is a decision for the design, not the requirements, stage. In addition, rather than aim for a narrow specification of the model to be implemented, the target should be a broad specification of the potential family of models. A program family approach, where commonalities are reused and variabilities are identified and systematically handled, is natural for scientific software [56]. As pointed out previously, at an abstract level, the modeler will know which governing conservation equations will need to be satisfied. The challenge is to know which simplifying assumptions are appropriate. This is where the "experimentation" by scientists comes in. If the assumptions are documented clearly, and explicit traceability is given to show what part of the model they influence, then changes can be made later, as understanding of the problem improves. Using knowledge from the field of SE, the documentation can be built with maintainability and reusability in mind.

This chapter shows how SE templates, rules and guidelines, which have been successful in other domains, can be adapted to handle rapid change and complexity. The document driven approach is first described (Section 1.2) and then illustrated via the example of software to model a solar water heating tank (Section 1.3). Justification (Section 1.4) for the document driven process is shown through a case study where legacy nuclear safety analysis code is re-documented, leading to the discovery of 27 issues in the original documentation. Further justification is given through a survey of statistical software for

psychology, which shows that quality is highest for projects that most closely follow a document driven approach.

1.2 A Document Driven Method

Table 1.2 shows the recommended documentation for a scientific software project. The documents are typical of what is suggested for scientific software certification, where certification consists of official recognition by an authority, or regulatory body, that the software is fit for its intended use. For instance, the Canadian Standards Association (CSA) requires a similar set of documents for quality assurance of scientific programs for nuclear power plants [8].

TABLE 1.2: Recommended Documentation

Problem Statement	Description of problem to be solved
Development Plan	Overview of development process/infrastructure
Requirements	Desired functions and qualities of the software
V&V Plan	Verification that all documentation artifacts, including the code, are internally correct. Validation, from an external viewpoint, that the right problem, or model, is being solved.
Design Specification	Documentation of how the requirements are to be realized, through both a software architecture and detailed design of modules and their interfaces
Code	Implementation of the design in code
V&V Report	Summary of the V&V efforts, including testing
User Manual	Instructions on installation, usage; worked examples

To achieve the qualities listed in Table 1.1, the documentation in Table 1.2 should have the following qualities: *complete*, *correct*, *consistent*, *modifiable*, *traceable*, *unambiguous*, and *verifiable*. All of these qualities are listed in the IEEE recommended practice for software requirements [22]. The IEEE guidelines are for requirements, but most qualities are relevant for all documentation artifacts. Another relevant quality, which is not on the IEEE list, is *abstract*. Requirements should state what is to be achieved, but be silent on how it is to be achieved. Abstraction is an important software development principle for dealing with complexity [17, p. 40]. Smith and Koothoor present further details on the qualities of documentation for scientific software [49].

The recommended rational process (Figure 1.1) is a variation on the waterfall model that is similar to the V-model. The steps proceed as for the waterfall model up to the requirements specification, but then detour to create the V&V plan, before renewing the waterfall. In the full V-model, each development phase has a corresponding test plan and report. The proposed

process does not go this far; one document summarizes the V&V plan at the crucial initial stage of development and provides an overview for the V&V of the other phases. In the traditional waterfall, test plans are in a later stage, but thinking about system tests early has the benefit that test cases are often more understandable than abstract requirements. System test cases should be considered at the same time as requirements; the tests themselves form an alternative, but incomplete, view of the requirements. Iteratively working between requirements and system tests builds confidence that the project is moving in the correct direction before making significant, and expensive, decisions.

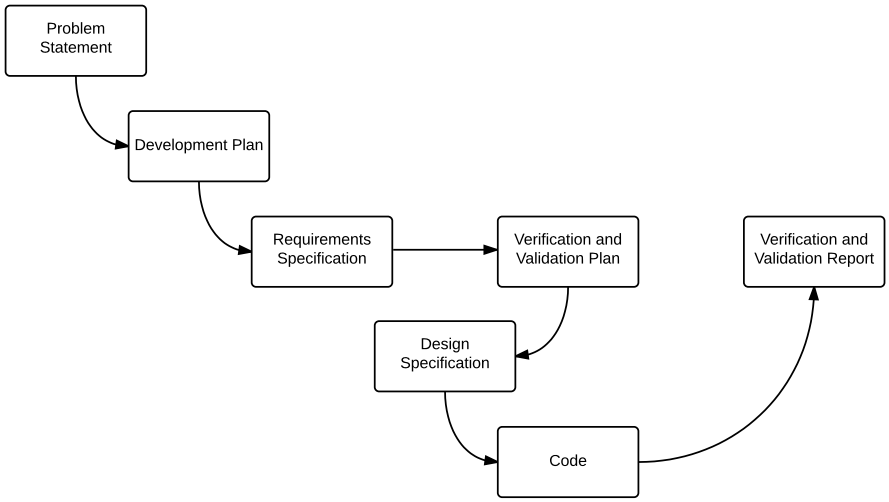

FIGURE 1.1: Overview of recommended process for documentation.

1.2.1 Problem Statement

A problem statement is a high level description of what the software hopes to achieve. Like a mission statement in a strategic plan [38, p. 22], the problem statement summarizes the primary purpose of the software, what it does, who the users are and what benefits the software provides. A problem statement should be abstract. That is, it should state what the mission is, but not how to achieve it. The length of a problem statement should usually be about half a page, or less. The seismology software Mineos [6] provides a good example of a problem statement, starting with "Mineos computes synthetic seismograms in a spherically symmetric non-rotating Earth by summing normal modes."

The problem statement's main impact on software quality is through improving reuse, since a clear statement positions the current work relative to similar software products. If the problem statement shows too much overlap

with existing products, the decision may be made to go in another direction. Moreover, the information in the problem statement might be enough to encourage future users and developers to adopt this product, rather than develop something new. The problem statement also improves quality, since it provides focus for subsequent work and documents.

1.2.2 Development Plan

The recommendations in this section imply a set of documents (Table 1.2), but the specific contents of these documents and the process that underlies them is not prescribed. As mentioned in Section 1.1, the external documentation follows a "faked" rational process, but the internal process can be anything that the developers desire. The development plan is where this internal process is specified. The specific parts of the plan should include the following:

- What documents will be created?
- What template, including rules and guidelines, will be followed?
- What internal process will be employed?
- What technology infrastructure (development support tools) will be used (see Section 1.2.8)?
- What are the coding standards?
- In the case of open source software, how does one contribute?

GRASS (Geographic Resources Analysis Support System) [18] provides a good example of community developed software that has a clear software development process and development support tool infrastructure [31]. The seismology software Earthworm [23] provides another solid example. The development plan for Earthworm distinguishes between three different categories of software: core, contributed and encapsulated. In addition, details are provided on the expected coding standards and on how to contribute to the project. Unfortunately, the requirements for contributing to Earthworm are sparse and the support tools seem to be limited to issue tracking.

The presence of a development plan immediately improves the qualities of visibility and transparency, since the development process is now defined. The plan also improves reproducibility, since it records the development and testing details. Depending on the choices made, the development support tools, such as version control and issue tracking, can have a direct impact on the quality of maintainability.

1.2.3 Software Requirements Specification (SRS)

The Software Requirements Specification (SRS) records the functionalities, expected performance, goals, context, design constraints, external interfaces

and other quality attributes of the software [22]. Writing an SRS generally starts with a template, which provides guidelines and rules for documenting the requirements. Several existing templates contain suggestions on how to avoid complications and how to achieve qualities such as verifiability, maintainability and reusability [15, 22, 35]. However, no template is universally accepted. For the current purpose, a good starting point is a template specifically designed for scientific software [54, 55], as illustrated in Figure 1.2. The recommended template is suitable for science, because of its hierarchical structure, which decomposes abstract goals to concrete instance models, with the support of data definitions, assumptions and terminology. The document's structure facilitates its maintenance and reuse [54], by using separation of concerns, abstraction and traceability, as discussed in Section 1.1.

FIGURE 1.2: SRS table of contents.

An SRS improves the software qualities listed in Table 1.1. For instance, usability is improved via an explicit statement of the expected user characteristics. Verifiability is improved because the SRS provides a standard against which correctness can be judged. The recommended template [54, 55] facili-

tates verification of the theory by systematically breaking the information into structured units, and using cross-referencing for traceability. An SRS also improves communication with stakeholders. To facilitate collaboration and team integration, the SRS captures the necessary knowledge in a self-contained document. If a standard template is adopted for scientific software, this would help with comparing between different projects and with reusing knowledge.

1.2.4 Verification and Validation (V&V) Plan and Report

Verification is not just important for the code. As shown by the IEEE Standard for Software Verification and Validation Plans [60, p. 412], V&V activities are recommended for each phase of the software development lifecycle. For instance, experts should verify that the requirements specification is reasonable with respect to the theoretical model, equations, assumptions etc. This verification activity is assisted by the use of a requirements template tailored to scientific software, as discussed in Section 1.2.3. Verification of the design and the code can potentially be improved by the use of Literate Programming, as discussed in Section 1.2.6. An important part of the verification plan is checking the traceability between documents to ensure that every requirement is addressed by the design, every module is tested, etc.

Developing test cases is particularly challenging for scientific software, since scientific problems typically lack a test oracle [27]. In the absence of a test oracle, the following techniques can be used to build system tests:

- Test cases can be selected that are a subset of the real problem for which a closed-form solution exists. When using this approach, confidence in the actual production code can only be built if it is the same code used for testing; that is, nothing is gained if a separate, simpler, program is written for testing the special cases.

- Verification test cases can be created by assuming a solution and using this to calculate the inputs. For instance, for a linear solver, if A and x are assumed, b can be calculated as $b = Ax$. Following this, $Ax^* = b$ can be solved and then x and x^*, which should theoretically be equal, can be compared. In the case of solving Partial Differential Equations (PDEs), this approach is called the Method of Manufactured Solutions [45].

- Most scientific software uses floating point arithmetic, but for testing purposes, the slower, but guaranteed correct, interval arithmetic [20] can be employed. The faster floating point algorithm can then be verified by ensuring that the calculated answers lie within the guaranteed bounds.

- Verification tests should include plans for convergence studies. The discretization used in the numerical algorithm should be decreased (usually halved) and the change in the solution assessed.

- Confidence can be built in a numerical algorithm by comparing the

results to another program that overlaps in functionality. If the test results do not agree, then one, possibly both of the programs is incorrect.

- The verification plan should also include test plans for nonfunctional requirements, like accuracy, performance and portability, if these are important implementation goals. Performance tests can be planned to describe how the software responds to changing inputs, such as problem size, condition number etc. Verification plans can include relative comparisons between the new implementation and competing products [52].

In addition to system test cases, the verification plan should outline other testing techniques that will be used to build confidence. For instance, the plan should describe how unit test cases will be selected, although the creation of the unit test cases will have to wait until the design is complete. The test plan should also identify what, if any, code coverage metrics will be used and what approach will be employed for automated testing. If other testing techniques, such as mutation testing, or fault testing [60], are to be employed, this should be included in the plan. In addition to testing, the verification plan should mention the plans for other techniques for verification, such as code walkthroughs, code inspections, correctness proofs etc. [17, 60].

Validation is also included in the V&V plan. For validation, the document should identify the experimental results for comparison to the simulated results. If the purpose of the code is a general purpose mathematical library there is no need for a separate validation phase.

Figure 1.3 shows the proposed template for capturing the V&V plan. The first two sections cover general and administrative information, including the composition of the testing team and important deadlines. The "Evaluation" section fleshes out the methods, tools and techniques while the "System Test Description" provides an example of a system test. In an actual V&V report, there would be multiple instances of this section, each corresponding to a different system test. In cases where validation tests are appropriate, each validation test would also follow this template.

The corresponding document for the V&V plan is the V&V report. Once the implementation and other documentation is complete, the V&V activities take place. The results are summarized in the report, with enough detail to convince a reader that all the planned activities were accomplished. The report should emphasize those changes that were made in a response to issues uncovered during verification and validation.

1.2.5 Design Specification

As for other documents, the design document serves three purposes: design, implementation and maintenance [21, p. 16]. The initial document is verified to ensure it provides a good start, then it is used for software creation, and later, when changes are needed, it is a reference for maintainers. The recommended

FIGURE 1.3: Proposed V&V plan table of contents.

approach to handle complexity in design is abstraction [60, p. 296]. For science, the inspiration for abstraction is the underlying mathematics.

The documentation should include a high level view of the software architecture, which divides the system into modules, and a low level view, which specifies the interfaces for the modules. A module is defined as a "work assignment given to a programmer or group of programmers" [40]. Wilson et al. advise modular design for scientific software [63], but are silent on the decomposition criterion. A good criterion is the principle of information hiding [39]. This principle supports design for change through the "secrets" of each module. As implied in Section 1.1, design for change is valuable for scientific software, where a certain amount of exploration is necessary.

The modular decomposition can be recorded in a Module Guide (MG) [40], which organizes the modules in a hierarchy by their secrets. Given his interest in embedded real time systems, the top-level decomposition from Parnas [40] includes a hardware hiding module. For scientific software on standard hard-

ware, with serial algorithms, simplification is usually possible, since the virtualization of the hardware will typically not have to be directly implemented by the programmer, being generally available via libraries, such as `stdio.io` in C. Further simplifications are available in scientific software, by taking advantage of the Input ⇒ Calculate ⇒ Output design pattern mentioned in Section 1.1. This pattern implies the presence of an input format hiding module, an input parameter data structure hiding module and an output format hiding module [21]. The bulk of the difference between designs comes through the modules dealing with calculations. Typical calculation modules hide data structures, algorithms and the governing physics. The application of the Parnas approach to scientific software has been illustrated by applying it to the example of a mesh generator [57].

Figure 1.4 shows the proposed template for the MG document. The document begins with an explicit statement of the anticipated, or likely, changes. These anticipated changes guide the design. If a likely change is required, then ideally only one module will need to be re-implemented. The "Module Decomposition" section lists the modules, organized by a hierarchy of related secrets. The top level decomposition of the hierarchy consists of hardware hiding, behavior hiding and software decision hiding modules [40]. For each module the secret it encapsulates and the service it provides are listed. Care is taken that each module lists only one secret and that secrets are in the form of nouns, not verbs. The example modules listed in the section of Figure 1.4 are typical of scientific software. The "Traceability Matrix" section shows how the anticipated changes map to modules, and how the requirements from the SRS map to modules. Section 1.3.2 describes an example MG, along with the uses hierarchy between modules.

The modular decomposition advocated here has much in common with Object Oriented (OO) design, which also emphasizes encapsulation. However, care must be taken with overusing OO languages, since a significant performance penalty is possible using dynamic dispatch, especially in an inner loop. Operator overloading should also be used with care, since the operator semantics may change depending on the type of its operands.

The MG alone does not provide enough information. Each module's interface needs to be designed and documented by showing the syntax and semantics of its access routines. This can be done in the Module Interface Specification (MIS) [21]. The MIS is less abstract than the architectural design. However, an MIS is still abstract, since it describes what the module will do, but not how to do it. The interfaces can be documented formally [14, 57] or informally. An informal presentation would use natural language, together with equations. The specification needs to clearly define all parameters, since an unclear description of the parameters is one cause of reusability issues for libraries [11]. To assist with interface design, one can take inspiration from the common design idioms for the structures of set, sequence and tuple [21, p. 82–83]. In addition, the designer should keep in mind the following interface quality criteria: consistent, essential, general, minimal and opaque [21, p. 83].

FIGURE 1.4: Proposed MG table of contents.

1.2.6 Code

Comments can improve understandability, since comments "aid the understanding of a program by briefly pointing out salient details or by providing a larger-scale view of the proceedings" [28]. Comments should not describe details of how an algorithm is implemented, but instead focus on what the algorithm does and the strategy behind it. Writing comments is one of the best practices identified for scientific software by Wilson et al. [63]. As said by Wilson et al., scientific software developers should aim to "write programs for people, not computers" and "[t]he best way to create and maintain reference documentation is to embed the documentation for a piece of software in that software" [63]. Literate Programming (LP) [30] is an approach that takes these ideas to their logical conclusion.

LP was introduced by Knuth [30]. The central idea is that "...instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do" [30, pg. 99]. When using LP, an algorithm is refined into smaller, simpler parts. Each of the parts is documented in an order that is natural for human comprehension, as opposed to the order used for compilation. In a literate program, documentation and code are maintained in one source.

The program is written as an interconnected "web" of "chunks" [30]. LP can be used as a strategy for improving verifiability, understandability and reproducibility for scientific software. One example of a commonly used LP tool is Sweave [32]. (This tool is discussed further in Section 1.4.1.) Other examples of LP in scientific software include a validated ODE solver [36] and a photorealistic renderer [43].

Figure 1.5 shows a sample of literate code documentation drawn from a program to analyze heat transfer in a nuclear reactor fuel pin [49]. The example shows how documentation and code can be interleaved. The excerpt begins with the mathematical formulae for the heat transfer coefficients and ends with the actual C code used for their calculation. By having the theory and the code side by side in this manner, a human reviewer will have an easier time verifying the code implements the theory. The excerpt also illustrates how LP automatically manages chunk numbering and referencing.

B.6.2 Computing h_c, h_g and T_S

Using this clad conductivity (k_c), we compute the heat transfer coefficient (h_c) and the gap conductance (h_g) as DD18 and DD19 of the SRS, respectively. That is,

$$h_c = \frac{2k_c h_b}{2k_c + \tau_c h_b}, \tag{B.23}$$

$$h_g = \frac{2k_c h_p}{2k_c + \tau_c h_p} \tag{B.24}$$

21 ⟨ Calculation of heat transfer coefficient (h_c) and the gap conductance (h_g) 21 ⟩ ≡

```
/* calculation of heat transfer coefficient */
*h_c = (2 * (*k_c) * (*h_b))/((2 * (*k_c)) + (*tau_c * (*h_b)));
/* calculation of gap conductance */
*h_g = (2 * (*k_c) * (*h_p))/((2 * (*k_c)) + (*tau_c * (*h_p)));
```

This code is used in chunks 15 and 60

FIGURE 1.5: Example literate code documentation.

1.2.7 User Manual

The presence of a user manual will have a direct impact on quality. The quality of usability in particular benefits from a user manual, especially if the manual includes a getting started tutorial and fully explained examples. A user manual also benefits installability, as long as it includes linear installation instructions. Advice for writing user manuals and technical instructions can be found in technical writing texts [61]. The seismology software Mineos [6] provides a good example of a user manual for scientific software.

1.2.8 Tool Support

Scientific software developers should use tools for issue tracking and version control. Issue tracking is considered a central quality assurance process [2]. Commercial issue trackers, such as Jira, are available, along with free tools such as, iTracker, Roundup, GitHub and Bugzilla [24]. For version control, frequently recommended tools are Subversion [44] and Git [33]. Version control tools can support reproducibility, since they are able to record development information as the project progresses. Davison [9], recommends more flexible and powerful automated reproducibility tools, such as Sumatra [10] and Madagascar [16]. Issue tracking and version control tools should be employed for all of the documents mentioned in the preceding sections.

Tool use for code documentation falls on a continuum between no tool use, all the way up to full LP (discussed in Section 1.2.6). In between these extremes there are code documentation assistants like Javadoc, Doxygen, Sphinx and publish for MATLAB®. These tools can be thought of as code first, then documentation. LP flips this around with documentation first, then code. Tools for LP include cweb, noweb, FunnelWeb and Sweave.

Tools also exist to make the testing phase easier. For functional testing, unit testing frameworks are popular. A unit testing framework has been developed for most programming languages. Examples include JUnit (for Java), Cppunit (for C++), CUnit (for C), FUnit (for FORTRAN), and PyUnit (for Python). For nonfunctional testing related to performance, one can use a profiler to identify the real bottlenecks in performance [63]. A powerful tool for dynamic analysis of code is Valgrind. The specific choices of tools for a given project should be documented in the V&V plan (Section 1.2.4).

1.3 Example: Solar Water Heating Tank

This section provides highlights of the documentation produced via a rational document driven process for a software program called SWHS. This program simulates a Solar Water Heating System (SWHS) incorporating Phase Change Material (PCM) [46]. Tanks are sometimes designed with PCM to reduce the tank size over a water only tank. Incorporating PCM reduces tank size since PCM stores thermal energy as latent heat, which allows higher thermal energy storage capacity per unit weight. Figure 1.6 provides a conceptual view of the heat flux (q) in the tank. The full set of documents, code and test cases, for the SWHS example can be found at: `https://github.com/smiths/swhs.git`.

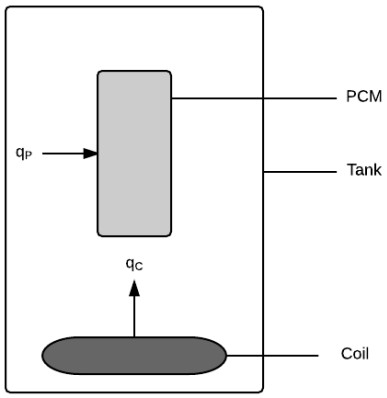

FIGURE 1.6: Solar water heating tank, with heat flux q_c from coil and q_P to the PCM.

1.3.1 Software Requirements Specification (SRS)

Figure 1.2 shows the table of contents for the SRS for SWHS. Although the SRS document is long (25 pages), this is in keeping with the knowledge capture goal. Someone, like a new undergraduate or graduate student, with a physics and mathematics background (as given in SRS Section 3.1, User Characteristics) will have all that they need to understand the software. The documentation is not written only for experts on heat transfer, but also for people that are trying to become experts. The documentation alone will not make someone an expert, but the intention is that it will provide enough detail, and enough pointers to additional information, that it can serve as a valuable learning, and later a reference, resource.

The table of contents shows how the problem is systematically decomposed into more concrete models. Specifically, the presentation starts with the high level problem goals (Figure 1.7), then the SRS provides the appropriate theoretical models to achieve the goals. The theoretical models are then refined into what are termed instance models, which provide the equations needed to solve the original problem. During this refinement from goals to theory to mathematical models, the scientist applies different assumptions, builds general definitions and creates data definitions. The template aids in documenting all the necessary information, since each section has to be considered. This facilitates achieving completeness by essentially providing a checklist. Besides requiring that section headings be filled in, the template also requires

that every equation either has a supporting external reference, or a derivation. Furthermore, for the SRS to be complete and consistent every symbol, general definition, data definition, and assumption needs to be used at least once.

The goal statements for SWHS, given in Figure 1.7, specify the target of the system. In keeping with the principle of abstraction, the goals are stated such that they describe many potential instances of the final program. As a consequence, the goals will be stable and reusable.

4.1.3 Goal Statements

Given the temperature of the coil, initial conditions for the temperature of the water and the PCM, and material properties, the goal statements are:

GS1: predict the water temperature over time

GS2: predict the PCM temperature over time

GS3: predict the change in the energy of the water over time

GS4: predict the change in the energy of the PCM over time

FIGURE 1.7: Goal statements for SWHS.

As mentioned in Section 1.1, scientists often need to experiment with their assumptions. For this reason, traceability information needs to be part of the assumptions, as shown in Figure 1.8. As the assumptions inevitably change, the analyst will know which portions of the documentation will potentially also need to change.

4.2.1 Assumptions

This section simplifies the original problem and helps in developing the theoretical model by filling in the missing information for the physical system. The numbers given in the square brackets refer to the theoretical model [T], general definition [GD], data definition [DD], instance model [IM], or likely change [LC], in which the respective assumption is used.

A1: The only form of energy that is relevant for this problem is thermal energy. All other forms of energy, such as mechanical energy, are assumed to be negligible [T1].

A2: All heat transfer coefficients are constant over time [GD1].

A3: The water in the tank is fully mixed, so the temperature is the same throughout the entire tank [GD2, DD2].

A4: The PCM has the same temperature throughout [GD2, DD2, LC1].

A5: Density of the water and PCM have not spatial variation; that is, they are each constant over their entire volume [GD2].

A6: Specific heat capacity of the water and PCM have no spatial variation; that is, they are each constant over their entire volume [GD2].

FIGURE 1.8: Sample assumptions for SWHS.

The abstract theoretical model for the conservation of thermal energy is presented in Figure 1.9. As discussed in Section 1.1, this conservation equation applies for many physical problems. For instance, this same model is used in the thermal analysis of a nuclear reactor fuel pin [49]. This is possible since the equation is written without reference to a specific coordinate system.

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 time varying heat transfer in a material of specific heat capacity C and density ρ, where $\mathbf{q}$ is the thermal flux vector, g is the volumetric heat generation, T is the temperature, t is time, 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).
Source	`http://www.efunda.com/formulae/heat_transfer/conduction/overview_cond.cfm`
Ref. By	GD2

FIGURE 1.9: Sample theoretical model.

T1 (Figure 1.9) can be simplified from an abstract theoretical model to a more problem specific General Definition (GD). Figure 1.10 shows one potential refinement (GD2), which can be derived using Assumptions A3–A6 (Figure 1.8). The specific details of the derivation are given in the full SRS on-line, but not reproduced here for space reasons. This restating of the conservation of energy is still abstract, since it applies for any control volume that satisfies the required assumptions. GD2 can in turn be further refined to specific (concrete) instanced models for predicting the temperature of the water and the PCM over time.

IM1 and IM2 provide the system of ODEs that needs to be solved to determine T_w and T_P. If a reader would prefer a bottom up approach, as opposed to the default top down organization of the original SRS, they can start their reading with the instance models and trace back to find any additional information that they require. IM2 is shown in Figure 1.11. Hyperlinks are included in the original documentation for easy navigation to the associated data definitions, assumptions and instance models.

To achieve a separation of concerns between the requirements and the design, the SRS is abstract, as discussed in Section 1.2. The governing ODEs are given, but not a solution algorithm. The focus is on "what" the software does, not "how" to do it. The numerical methods are left to the design document. This approach facilitates change, since a new numerical algorithm requires no changes to the SRS.

Number	GD2
Label	**Simplified rate of change of temperature**
Equation	$mC\frac{dT}{dt} = q_{\text{in}}A_{\text{in}} - q_{\text{out}}A_{\text{out}} + gV$
Description	The basic equation governing the rate of change of temperature, for a given volume V, with time. m is the mass (kg). C is the specific heat capacity ($\text{J kg}^{-1}\,^{\circ}\text{C}^{-1}$). T is the temperature (°C) and t is the time (s). q_{in} and q_{out} are the in and out heat transfer rates, respectively (W m^{-2}). A_{in} and A_{out} are the surface areas over which the heat is being transferred in and out, respectively (m^2). g is the volumetric heat generated (W m^{-3}). V is the volume (m^3).
Ref. By	IM1, IM2

FIGURE 1.10: Sample general definition.

If an expert reviewer is asked to "sign off" on the documentation, he or she should find an explanation/justification for every symbol/equation/definition. This is why IM2 not only shows the equation for energy balance to find T_P, but also the derivation of the equation. (The derivation is not reproduced here for space reasons, but it can be found at `https://github.com/smiths/swhs.git`.)

Table 1.3 shows an excerpt for the table summarizing the input variables for SWHS. With the goal of knowledge capture in mind, this table includes constraints on the input values, along with data on typical values. When new users are learning software, they often do not have a feel for the range and magnitude of the inputs. This table is intended to help them. It also provides a starting point for later testing of the software. The uncertainty information is included to capture expert knowledge and to facilitate later uncertainty quantification analysis.

1.3.2 Design Specification

The modular decomposition for SWHS is recorded in a Module Guide (MG), as discussed in Section 1.2.5. The table of contents of the MG for SWHS is similar to that shown in Figure 1.4, but with a few different modules. The specific modules for SWHS are shown in Figure 1.12, which summarizes the uses relation between the modules, where module A uses module B if a correct execution of B may be necessary for A to complete the task described

Number	IM2
Label	**Energy balance on PCM to find T_P**
Input	m_P, C_P^S, C_P^L, h_P, A_P, t_{final}, T_{init}, T_{melt}^P, $T_W(t)$ from IM1 The input is constrained so that $T_{\text{init}} < T_{\text{melt}}^P$ (A13)
Output	$T_P(t)$, $0 \leq t \leq t_{\text{final}}$, with initial conditions, $T_W(0) = T_P(0) = T_{\text{init}}$ (A12), and $T_W(t)$ from IM1, such that the following governing ODE is satisfied. The specific ODE depends on T_P as follows: $\frac{dT_P}{dt} = \begin{cases} \frac{dT_P}{dt} = \frac{1}{\tau_P^S}(T_W(t) - T_P(t)) & \text{if } T_P < T_{\text{melt}}^P \\ \frac{dT_P}{dt} = \frac{1}{\tau_P^L}(T_W(t) - T_P(t)) & \text{if } T_P > T_{\text{melt}}^P \\ 0 & \text{if } T_P = T_{\text{melt}}^P \text{ and } 0 < \phi < 1 \end{cases}$ The temperature remains constant at T_{melt}^P, even with the heating (or cooling), until the phase change has occurred for all of the material; that is as long as $0 < \phi < 1$. ϕ (from DD4) is determined as part of the heat energy in the PCM, as given in IM4 $t_{\text{melt}}^{\text{init}}$, the temperature at which melting begins. $t_{\text{melt}}^{\text{final}}$, the temperature at which melting ends.
Description	T_W is water temperature (°C). T_P is the PCM temperature (°C). $\tau_P^S = \frac{m_P C_P^S}{h_P A_P}$ is a constant (s). $\tau_P^L = \frac{m_P C_P^L}{h_P A_P}$ is a constant (s).
Sources	[4]
Ref. By	IM1, IM4

FIGURE 1.11: Sample instance model.

in its specification. The relation is hierarchical, which means that subsets of the design can be independently implemented and tested. Some specific points about the MG are as follows:

- Likely changes for SWHS include "the format of the initial input data" and the "algorithm used for the ODE solver." The likely changes are the basis on which the modules are defined.
- One straightforward module is the Input Format Module (M2). This module hides the format of the input, as discussed generically in Section 1.2.5. It knows the structure of the input file, so that no other module needs to know this information. The service that the Input Format Module provides is to read the input data and then modify the state of the Input Parameters Module (M3) so that it holds all of the required information.

TABLE 1.3: Excerpt from Table of Input Variables for SWHS

Var	Physical Constraints	Software Constraints	Typical Value	Uncertainty
L	$L > 0$	$L_{\min} \leq L \leq L_{\max}$	1.5 m	10%
D	$D > 0$	$\frac{D}{L}_{\min} \leq \frac{D}{L} \leq \frac{D}{L}_{\max}$	0.412 m	10%
V_P	$V_P > 0$ (*)	$V_P \geq \text{minfract} \cdot V_{\text{tank}}(D, L)$	0.05 m^3	10%
	$V_P < V_{\text{tank}}(D, L)$			
A_P	$A_P > 0$ (*)	$V_P \leq A_P \leq \frac{2}{h_{\min}} V_P$ (#)	1.2 m^2	10%
ρ_P	$\rho_P > 0$	$\rho_P^{\min} < \rho_P < \rho_P^{\max}$	1007 kg/m^3	10%
T^P_{melt}	$0 < T^P_{\text{melt}} < T_C$		44.2 °C	10%

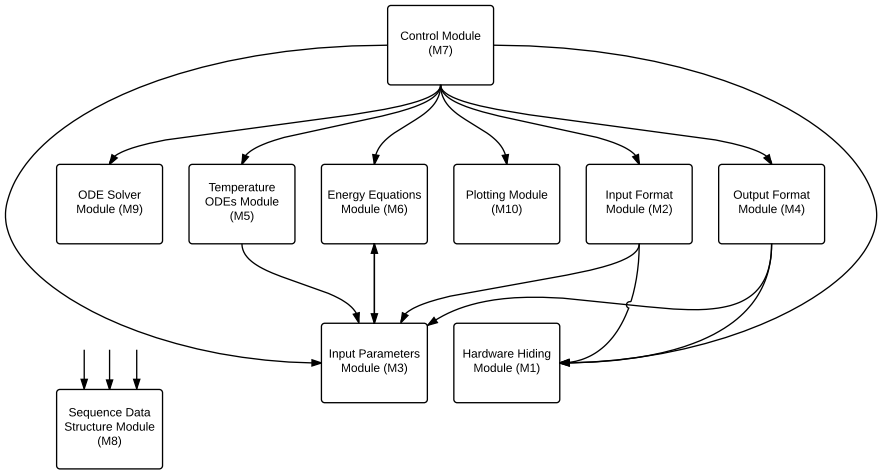

FIGURE 1.12: Uses hierarchy among modules.

- Several of the modules that are documented, such as the Sequence Data Structure Module (M8) and the ODE Solver Module (M9), are already available in Matlab, which is the selected implementation environment for SWHS. These modules are still explicitly included in the design, with a notation that indicates that they will be implemented by Matlab. They are included so that if the implementation environment is later changed, the developer will know that they need to provide these modules.

- The MG shows the traceability matrix between the modules and the SRS requirements. This traceability increases confidence that the design is complete because each requirement maps to a module, and each module maps to at least one requirement.

1.4 Justification

Part of the justification for the document driven approach presented in this chapter is an appeal to the value of a systematic, rigorous, engineering approach. This approach has been successful in other domains, so it stands to reason that it should be successful for scientific software. The example of the solar water heating tank provides partial support for this, since the documentation and code were positively reviewed by a mechanical engineer. Although only providing anecdotal evidence in support of the documentation, the reviewer liked the explicit assumptions; the careful description of names, nomenclature and units; and, the explicit planning for change in the design. The reviewer thought that the documentation captured knowledge that would facilitate new project members quickly getting up to speed. The reviewer's main concern was the large amount of documentation for such a relatively simple, and non-safety critical, problem. This concern can be mitigated by the following observations: i) the solar water heating example was intentionally treated more seriously than the problem perhaps deserves, so that a relatively small, but still non-trivial example, could be used to illustrate the methods proposed in this paper; and, ii) if the community recognizes the value of rational documentation, then tool support will follow to reduce the documentation burden. This last point is explored further in the Concluding Remarks (Section 1.5).

Justification by appeals to success in other domains, and by positive comments from a review of SWHS, are not entirely satisfying. Maybe there really is something about science that makes it different from other domains? The research work presented below further justifies that this is not the case.

1.4.1 Comparison between CRAN and Other Communities

The value of documentation and a structured process is illustrated by a survey of statistical software for psychology [50, 51]. The survey compares the quality of statistical software when it is developed using an ad hoc process versus employing the CRAN (Comprehensive R Archive Network [7]) process.

Thirty software tools were reviewed and ranked with respect to their adherence to software engineering best practices. For the surveyed software, R packages clearly performed better than the other categories for qualities related to development, such as maintainability, reusability, understandability and visibility. Commercial software, for which the development process was unknown, provided better usability, but did not show as much evidence of verifiability. With respect to usability, a good CRAN example is mokken [59].

The overall high ranking of R packages stems largely from their use of Rd, Sweave, R CMD check and the CRAN Repository Policy. The policy and support tools mean that even a single developer project can be sufficiently well

documented and developed to be used by others. A small research project usually does not have the resources for an extensive development infrastructure and process. By enforcing rules for structured development and documentation, CRAN is able to improve the quality of scientific software.

1.4.2 Nuclear Safety Analysis Software Case Study

To study the impact of a document driven process on scientific software, a case study was performed on legacy software used for thermal analysis of a fuel pin in a nuclear reactor [49, 53]. The legacy code and theory manual were compared to a redeveloped version of the code and documentation using the rational process described in this paper. The redeveloped documentation focused on a single module, the thermal analysis module, and emphasized the SRS and the use of LP. The case study is considered representative of many other scientific programs.

Highly qualified domain experts produced the theory manual for the original software. Their goal was to fully explain the theory for the purpose of QA. The documentation followed the usual format for scientific journals or technical report. Even with an understanding of the importance of the original documentation, the redeveloped version uncovered 27 issues in the previous documentation, ranging from trivial to substantive. Although no errors were uncovered in the code itself, the original documentation had problems with incompleteness, ambiguity, inconsistency, verifiability, modifiability, traceability and abstraction.

The redeveloped code used LP for documenting the numerical algorithms and the code, in what was termed the Literate Programmer's Manual (LPM). An excerpt from the LPM is shown in Figure 1.5. This excerpt shows that the LPM was developed with explicit traceability to the SRS. The traceability between the theory, numerical algorithms and implementation, facilitates achieving completeness and consistency, and simplifies the process of verification and the associated certification. The case study shows that it is not enough to say that a document should be complete, consistent, correct and traceable; practitioners need guidance on how to achieve these qualities.

The case study highlights how a document driven process can improve verifiability. To begin with, verification is only possible with an explicit statement of the requirements, or, in the terminology of Table 1.1, a list of the "right equations." The case study found that, due to inconsistent use of symbols for heat transfer coefficients, two equations for thermal resistance in the original theory manual were actually the "wrong equations." A more systematic process and explicit traceability between theory and code, would have caught these mistakes. The use of LP was also shown to improve the quality of verifiability, since all the information for the implementation is given, including the details of the numerical algorithms, solution techniques, assumptions and the program flow. The understandability of LP is a great benefit for code reading, which is a key activity for scientists verifying their code [25].

Although some of the problems in the original documentation for the case study would likely have been found with any effort to redo the documentation, the rational process builds confidence that the methodology itself improves quality. The proposed SRS template assisted in systematically developing the requirements. The template helped in achieving completeness by acting as a checklist. Since the template was developed following the principle of separation of concerns, each section could be dealt with individually, and the details for the document could be developed by refining from goals to instanced models. The proposed template provides guidelines for documenting the requirements by suggesting an order for filling in the details. This reduces the chances of missing information. Verification of the documentation involves checking that every symbol is defined; that every symbol is used at least once; that every equation either has a derivation, or a citation to its source; that every general definition, data definition and assumption is used by at least one other component of the document; and, that every line of code either traces back to a description of the numerical algorithm (in the LPM), or to a data definition, or to an instance model, or to an assumption, or to a value from the auxiliary constants table in the SRS.

In all software projects, there is a danger of the code and documentation getting out of sync, which seems to have been a problem in the legacy software. LP, together with a rigorous change management policy, mitigates this danger. LPM develops the code and design in the same document, while maintaining traceability between them, and back to the SRS. As changes are proposed, their impact can be determined and assessed.

1.5 Concluding Remarks

This chapter has shown the feasibility of a rational document driven process for scientific software. In the past, scientific software has likely suffered from the vicious cycle where documentation quality is continually eroded [42]. Since software documentation is disliked by almost everyone, the documentation that is typically produced is of poor quality. The reduced quality leads to reduced usage, and the reduced usage in turn leads to a reduction in both resources and motivation. The reduced resources and motivation means further degradation in quality, and the vicious cycle repeats. Following a document driven process, like that described here, can potentially end this vicious cycle.

The approach recommended in this paper is to produce the full suite of software artifacts described in Section 1.2. However, this may be a daunting task to start out with, especially for projects that begin with a small scope. A practical starting point is to adopt tools wherever possible to simplify and improve the development process. In particular, a version control system is an important building block [62]. Ideally developers should adopt a full web

solution, like GitHub, or SourceForge, which provide documentation and code management, along with issue tracking. This approach provides the advantage that the product website can be designed for maximum visibility [31]. Moreover, the project can gradually grow into the use of the available tools as the need arises. For code documentation, developers should initially use a tool like Doxygen, since this enforces consistency and produces documentation so that other developers can more easily navigate the source code.

Although requirements are important, a more natural starting point for many developers seems to be test cases, likely because test cases are less abstract than requirements. To ease into a rational process, a project might begin the development process by writing test cases, which in a sense form the initial requirements for the project. If an infrastructure for automated testing is created early in a project, this can help improve verification and validation efforts going forward.

One potential shortcoming of the proposed approach is its reliance on human beings. Following the documentation templates in Section 1.2, and keeping all software artifacts in sync, should produce high quality software, but this places a burden on the developers and reviewers to pay attention to many details. Future work is planned to reduce this burden. Additional tool support, such as the Drasil framework [58], can be incorporated into the process. The fundamental task for Drasil is knowledge capture, so that this knowledge can be used to generate the SRS, V&V plan, MG, code, etc. In Drasil, each individual piece of knowledge is a named chunk, as in LP. Chunks are assembled using a recipe and a generator then interprets recipes to produce the desired document. With Drasil, the task of reusing knowledge, managing change and maintaining traceability sits with the computer, which is much better suited to these tasks than the typical human being.

Just as a compiler can check that all variables have been initialized, the new tools can check the SRS for completeness and consistency and verify that rules, like the one that all symbols are defined, are enforced. Code generation techniques for scientific software [3, 4, 29] can be used to generalize the idea of LP from the code to cover all software artifacts. A Domain Specific Language (DSL) can be designed for capturing mathematical knowledge for families of scientific software. For instance, any repetition between documents can automatically be generated, rather than relying on a manual process. Ideally, code generation can be used to transform portions of the requirements directly into code. Furthermore, generation techniques may be used to generate documentation to suit the needs of a particular user. For instance, the details on the proof or derivation of equations can be removed for viewers using the software for maintenance purposes, but added back in for reviewers verifying the mathematical model. The user can specify the "recipe" for their required documentation using the developed DSL.

Tool support will make the process easier, but practitioners should not wait. The document driven methods as presented here are feasible today and should be employed now to facilitate high quality scientific software. If an

approach such as that described in this paper becomes standard, then the work load will be reduced over time as documentation is reused and as practitioners become familiar with the templates, rules, and guidelines.

Bibliography

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

[2] Arne Beckhause, Dirk Neumann, and Lars Karg. The impact of communcation structure on issue tracking efficiency at a large business software vendor. *Issues in Information Systems*, X(2):316–323, 2009.

[3] Jacques Carette. Gaussian elimination: A case study in efficient genericity with MetaOCaml. *Science of Computer Programming*, 62(1):3–24, 2006.

[4] Jacques Carette, Mustafa ElSheikh, and W. Spencer Smith. A generative geometric kernel. In *ACM SIGPLAN 2011 Workshop on Partial Evaluation and Program Manipulation (PEPM'11)*, pages 53–62, January 2011.

[5] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. 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*, pages 550–559, Washington, DC, USA, 2007. IEEE Computer Society.

[6] CIG. Mineos. `http://geodynamics.org/cig/software/mineos/`, March 2015.

[7] CRAN. The comprehensive R archive network. `https://cran.r-project.org/`, 2014.

[8] CSA. Quality assurance of analytical, scientific, and design computer programs for nuclear power plants. Technical Report N286.7-99, Canadian Standards Association, 178 Rexdale Blvd. Etobicoke, Ontario, Canada M9W 1R3, 1999.

[9] Andrew P. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science & Engineering*, 14(4):48–56, July-Aug 2012.

[10] Andrew P. Davison, M. Mattioni, D. Samarkanov, and B. Teleńczuk. Sumatra: A toolkit for reproducible research. In V. Stodden, F. Leisch,

and R.D. Peng, editors, *Implementing Reproducible Research*, pages 57–79. Chapman & Hall/CRC, Boca Raton, FL, March 2014.

[11] Paul F. Dubois. Designing scientific components. *Computing in Science and Engineering*, 4(5):84–90, September 2002.

[12] Paul F. Dubois. Maintaining correctness in scientific programs. *Computing in Science & Engineering*, 7(3):80–85, May-June 2005.

[13] Steve M. Easterbrook and Timothy C. Johns. Engineering the software for understanding climate change. *IEEE Des. Test*, 11(6):65–74, 2009.

[14] Ahmed H. ElSheikh, W. Spencer Smith, and Samir E. Chidiac. Semi-formal design of reliable mesh generation systems. *Advances in Engineering Software*, 35(12):827–841, 2004.

[15] ESA. ESA software engineering standards, PSS-05-0 issue 2. Technical report, European Space Agency, February 1991.

[16] Sergey Fomel. Madagascar Project Main Page. `http://www.ahay.org/wiki/Main_Page`, 2014.

[17] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

[18] GRASS Development Team. GRASS GIS bringing advanced geospatial technologies to the world. `http://grass.osgeo.org/`, 2014.

[19] Michael Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill Publishing Company, New York, NY, USA, 2nd edition, 2002.

[20] Timothy Hickey, Qun Ju, and Maarten H. Van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48(5):1038–1068, September 2001.

[21] Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995.

[22] IEEE. *Recommended Practice for Software Requirements Specifications, IEEE Std. 830*. IEEE, 1998.

[23] ISTI. Earthworm software standards. `http://www.earthwormcentral.org/documentation2/PROGRAMMER/SoftwareStandards.html`, September 2013.

[24] Jeffrey N Johnson and Paul F Dubois. Issue tracking. *Computing in Science & Engineering*, 5(6):71–77, 2003.

[25] Diane Kelly. 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, pages 299–310, Riverton, NJ, USA, 2013. IBM Corp.

[26] Diane Kelly. Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. *Journal of Systems and Software*, 109:50–61, 2015.

[27] Diane F. Kelly, W. Spencer Smith, and Nicholas Meng. Software engineering for scientists. *Computing in Science & Engineering*, 13(5):7–11, October 2011.

[28] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley Professional, Reading, MA, 1999.

[29] Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 249–258, New York, NY, USA, 2004. ACM.

[30] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Center for the Study of Language and Information, 1992.

[31] Adam Lazzarato, Spencer Smith, and Jacques Carette. State of the practice for remote sensing software. Technical Report CAS-15-03-SS, McMaster University, January 2015. 47 pp.

[32] Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, Heidelberg, 2002. ISBN 3-7908-1517-9.

[33] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, Inc., 2012.

[34] Thomas Maibaum and Alan Wassyng. A product-focused approach to software certification. *IEEE Computer*, 41(2):91–93, 2008.

[35] NASA. Software requirements DID, SMAP-DID-P200-SW, release 4.3. Technical report, National Aeronautics and Space Agency, 1989.

[36] Nedialko S. Nedialkov. Implementing a Rigorous ODE Solver through Literate Programming. Technical Report CAS-10-02-NN, Department of Computing and Software, McMaster University, 2010.

[37] Suely Oliveira and David E. Stewart. *Writing Scientific Software: A Guide to Good Style*. Cambridge University Press, New York, NY, USA, 2006.

[38] Linda Parker Gates. Strategic planning with critical success factors and future scenarios: An integrated strategic planning framework. Technical Report CMU/SEI-2010-TR-037, Software Engineering Institute, Carnegie-Mellon University, November 2010.

[39] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

[40] David L. Parnas, P. C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.

[41] David L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.

[42] David Lorge Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering.*, pages 125–148, 2010.

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

[44] Michael Pilato. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[45] Patrick J. Roache. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque, New Mexico, 1998.

[46] Padideh Sarafraz. Thermal optimization of flat plate PCM capsules in natural convection solar water heating systems. Master's thesis, McMaster University, Hamilton, ON, Canada, 2014. http://hdl.handle.net/11375/14128.

[47] Judith Segal. When software engineers met research scientists: A case study. *Empirical Software Engineering*, 10(4):517–536, October 2005.

[48] Judith Segal and Chris Morris. Developing scientific software. *IEEE Software*, 25(4):18–20, July/August 2008.

[49] Spencer Smith and Nirmitha Koothoor. A document driven method for certifying scientific computing software used in nuclear safety analysis. *Nuclear Engineering and Technology*, Accepted, October 2015. 42 pp.

[50] Spencer Smith, Yue Sun, and Jacques Carette. Comparing psychometrics software development between CRAN and other communities. Technical Report CAS-15-01-SS, McMaster University, January 2015. 43 pp.

[51] Spencer Smith, Yue Sun, and Jacques Carette. Statistical software for psychology: Comparing development practices between CRAN and other communities. *Software Quality Journal*, Submitted December 2015. 33 pp.

[52] W. Spencer Smith. Systematic development of requirements documentation for general purpose scientific computing software. In *Proceedings of the 14th IEEE International Requirements Engineering Conference, RE 2006*, pages 209–218, Minneapolis / St. Paul, Minnesota, 2006.

[53] W. Spencer Smith, Nirmitha Koothoor, and Nedialko Nedialkov. Document driven certification of computational science and engineering software. In *Proceedings of the First International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCE)*, November 2013. 8 pp.

[54] W. Spencer Smith and Lei Lai. A new requirements template for scientific computing. In J. Ralyté, P. Ågerfalk, and N. Kraiem, editors, *Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP'05*, pages 107–121, Paris, France, 2005. In conjunction with 13th IEEE International Requirements Engineering Conference.

[55] W. Spencer Smith, Lei Lai, and Ridha Khedri. Requirements analysis for engineering computation: A systematic approach for improving software reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation*, 13(1):83–107, February 2007.

[56] W. Spencer Smith, John McCutchan, and Fang Cao. Program families in scientific computing. In Jonathan Sprinkle, Jeff Gray, Matti Rossi, and Juha-Pekka Tolvanen, editors, *7th OOPSLA Workshop on Domain Specific Modelling (DSM'07)*, pages 39–47, Montréal, Québec, October 2007.

[57] W. Spencer Smith and Wen Yu. A document driven methodology for improving the quality of a parallel mesh generation toolbox. *Advances in Engineering Software*, 40(11):1155–1167, November 2009.

[58] Daniel Szymczak, Spencer Smith, and Jacques Carette. Position paper: A knowledge-based approach to scientific software development. In *Proceedings of SE4Science'16*, United States, May 16 2016. In conjunction with ICSE 2016. 4 pp.

[59] L. Andries van der Ark. *mokken: Mokken Scale Analysis in R*, 2013. R package version 2.7.5.

[60] Hans van Vliet. *Software Engineering (2nd ed.): Principles and Practice*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[61] Judith S. VanAlstyne. *Professional and Technical Writing Strategies*. Pearson Prentice Hall, Upper Saddle River, New Jersey, sixth edition, 2005.

[62] Gregory V. Wilson. Where's the real bottleneck in scientific computing? Scientists would do well to pick some tools widely used in the software industry. *American Scientist*, 94(1), 2006.

[63] Gregory V. Wilson, D.A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H.D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumblet, Ben Waugh, Ethan P. White, and Paul Wilson. Best practices for scientific computing. *CoRR*, abs/1210.0530, 2013.