4
Software engineering

Software engineering encompasses the tools and methods for defining requirements for, designing, programming, testing, and managing software. It consists of monitoring and controlling both the software processes and the software products to ensure reliability. Software engineering was developed primarily from within the computer science community, and its use is essential for large software development projects and for high-assurance software systems such as those for aircraft control systems, nuclear power plants, and medical devices (e.g., pacemakers).

The reader may wonder at this point why a book on verification and validation in scientific computing includes a chapter on software engineering. The reason is that software engineering is critical for the efficient and reliable development of scientific computing software. Failure to perform good software engineering throughout the life cycle of a scientific computing code can result in much more additional code verification testing and debugging. Furthermore, it is extremely difficult to estimate the effect of *unknown* software defects on a scientific computing prediction (e.g., see Knupp *et al.*, 2007). Since this effect is so difficult to quantify, it is prudent to minimize the introduction of software defects through good software engineering practices.

Software engineers will no doubt argue that we have it backwards: *code verification* is really just a part of the software engineering process known as *software verification and validation*. While this is technically true, the argument for instead including software engineering as a part of code verification can be made as follows. Recall that we have defined scientific computing as the approximate solution to mathematical models consisting of partial differential or integral equations. The "correct" answer that should result from running a scientific computing code on any given problem is therefore not known: it will depend on the chosen discretization scheme, the chosen mesh (both its resolution and quality), the iterative convergence tolerance, the machine round-off error, etc. Thus special procedures must be used to test scientific computing software for coding mistakes and other problems that do not need to be considered for more general software. The central role of code verification in establishing the correctness of scientific computing software justifies our inclusion of software engineering as part of the code verification process. Regardless of the relation between software engineering and code verification, they are both important factors in developing and maintaining reliable scientific computing codes.

Computational scientists and engineers generally receive no formal training in modern software engineering practices. Our own search of the software engineering literature found a large number of contributions in various textbooks, on the web, and in scientific articles – mostly dominated by software engineering practices and processes that do not consider some of the unique aspects of scientific software. For example, most software engineering practices are driven by the fact that data organization and access is the primary factor in the performance efficiency of the software, whereas in scientific computing the speed of performing floating point operations is often the overriding factor. The goal of this chapter is to provide a brief overview of recommended software engineering practices for scientific computing. The bulk of this chapter can be applied to all scientific computing software projects, large or small, whereas the final section addresses additional software engineering practices that are recommended for large software projects.

Software engineering is an enormously broad subject which has been addressed by numerous books (e.g., Sommerville, 2004; McConnell, 2004; Pressman, 2005) as well as a broad array of content on the World Wide Web (e.g., SWEBOK, 2004; Eddins, 2006; Wilson, 2009). In 1993, a comprehensive effort was initiated by the Institute of Electrical and Electronics Engineers (IEEE) Computer Society to "establish the appropriate set(s) of criteria and norms for professional practice of software engineering upon which industrial decisions, professional certification, and educational curricula can be based" (SWEBOK, 2004). The resulting book was published in 2004 and divides software engineering into ten knowledge areas which comprise the Software Engineering Body of Knowledge (SWEBOK). In addition, there have been several recent workshops which address software engineering issues specifically for scientific computing (SE-CSE 2008, 2009) and high-performance computing (e.g., SE-HPC, 2004). In this chapter we will cover in detail the following software engineering topics: software development, version control, software testing, software quality and reliability, software requirements, and software management. An abbreviated discussion of many of the topics presented in this chapter can be found in Roy (2009).

4.1 Software development

Software development encompasses the design, construction, and maintenance of software. While software testing should also be an integral part of the software development process, we will defer a detailed discussion of software testing until a later section.

4.1.1 Software process models

A software process is an activity that leads to the creation or modification of software products. There are three main software process models: the waterfall model, the iterative and incremental development model, and component-based software engineering (Sommerville, 2004). In the traditional *waterfall model*, the various aspects of the software development process (requirements specification, architectural design, programming, testing, etc.) are decomposed into separate phases, with each phase beginning only after the previous phase is completed. In response to criticisms of the waterfall software development model, a competing approach called *iterative and incremental development* (also called the spiral model) was proposed. This iterative, or evolutionary, development model is based on the idea of interweaving each of the steps in the software development process, thus allowing customer feedback early in the development process through software prototypes which may initially have only limited capabilities. These software prototypes are then refined based on the customer input, resulting in software with increasing capability. A third model, *component-based software engineering*, can be used when a large number of reusable components are available, but often has only limited applicability to scientific computing (e.g., for linear solver libraries or parallel message passing libraries). Most modern software development models, such as the rational unified process (Sommerville, 2004) and agile software development (discussed later in this section), are based on the iterative and incremental development model.

4.1.2 Architectural design

Software architectural design is the process of identifying software sub-systems and their interfaces before any programming is done (Sommerville, 2004). The primary products of architectural design are usually documents (flowcharts, pseudocode, etc.) which describe the software subsystems and their structure. A software subsystem is defined as a subset of the full software system that does not interact with other subsystems. Each software subsystem is made up of components, which are subsets of the full system which interact with other components. Components may be based on a procedural design (subroutines, functions, etc.) or an object-oriented design, and both approaches are discussed in more detail in the next section.

4.1.3 Programming languages

There are a variety of factors to consider when choosing a programming language. The two main programming paradigms used in scientific computing are procedural programming and object-oriented programming. *Procedural programming* relies on calls to different procedures (routines, subroutines, methods, or functions) to execute a series of sequential steps in a given programming task. A significant advantage of procedural programming is that it is modular, i.e., it allows for reuse of procedures when tasks must be performed multiple times. In *object-oriented programming*, the program is decomposed into objects which interact with each other through the sending and receiving of messages. Objects typically make use of private data which can only be accessed through that object, thus providing a level of independence to the objects. This independence makes it easier to modify a given object without impacting other parts of the code. Object-oriented programming also allows for the reusability of components across the software system.

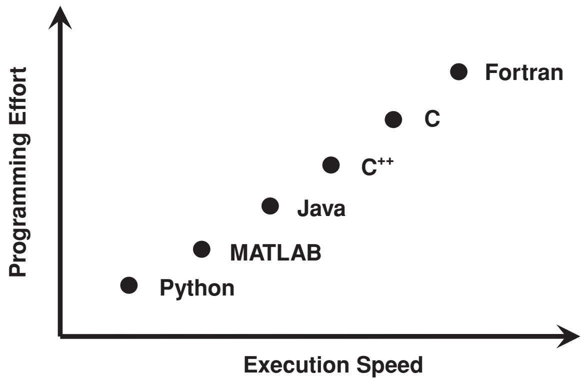

Figure 4.1 *Qualitative* example of programming effort versus execution speed (adapted from Wilson, 2009).

Most modern, higher-level programming languages used for scientific computing support both procedural and object-oriented programming. Programming languages that are primarily procedural in nature include BASIC, C, Fortran, MATLAB, Pascal, and Perl, while those that are primarily object-oriented include C^{++}, Java, and Python. Procedural programming is often used when mathematical computations drive the design, whereas object-oriented programming is preferred when the problem is driven by complex data relationships.

Low level computing languages such as machine language and assembly language (often found in simple electronic devices) execute extremely fast, but require additional time and effort during the programming and debugging phases. One factor to consider is that high-level languages, which often make use of more natural language syntax and varying levels of programming abstraction, have the advantage of making programming complex software projects easier, but generally will not execute as fast as a lower-level programming language. A qualitative comparison of selected programming languages is shown in Figure 4.1 which compares programming effort to execution speed. In scientific computing, the higher-level programming languages such as Python, MATLAB, and Java are ideal for small projects and prototyping, while production-level codes are usually programmed in C, C^{++}, or Fortran due to the faster execution speeds.

Another factor to consider when choosing a programming language is the impact on the software defect rate and the subsequent maintenance costs (Fenton and Pfleeger, 1997). Here we define a *software defect* as an error in the software that could potentially lead to software failure (e.g., incorrect result produced, premature program termination) and the *software defect rate* as the number of defects per 1000 lines of executable code. Evidence suggests that the software defect rate is at best weakly dependent on the choice of programming language (Hatton, 1997a). However, Hatton (1996) found that defects in object-oriented languages can be more expensive to find and fix, possibly by as much as a factor of three. The choice of compiler and diagnostic/debugging tool can also have a significant impact on the software defect rate as well as the overall code development productivity.

Standards for programming languages are generally developed by a costly and complex process. However, most programming language standards still contain coding constructs that are prone to producing software failures. These failure-prone constructs can arise in a number of different ways (Hatton, 1997a) including simple oversight by the standards process, lack of agreement on the content of the standards, because the decision was explicitly made to retain the functionality provided by the construct, or because of errors in the programming language standards documentation. In some cases, less fault-prone subsets of a programming language exist which reduce or eliminate the presence of the dangerous coding constructs. One example of a safe subset for the C programming language is Safer C (Hatton, 1995).

4.1.4 Agile programming

Most software development processes call for the requirements specification, design, implementation, and testing of the software to be performed sequentially. In this approach, changes to requirements can be costly and lead to extensive delays since they will impact the entire software development process. One notable exception is *agile programming* (also referred to as rapid software development, see agilemanifesto.org/), where requirements specification, design, implementation, and testing occur simultaneously (Sommerville, 2004).

Agile programming is iterative in nature, and the main goal is to develop useful software quickly. Some features of agile programming methods include:

- concurrency of development activities,
- minimal or automatic design documentation,
- only high-priority user requirements specified up front,
- significant user involvement in the development process, and
- incremental software development.

One of the intended advantages of agile programming is to provide software delivery (although initially with reduced capability) that allows for user involvement and feedback during the software design process and not just after the final software product has been delivered. Agile programming methods are good for small and medium-size software development efforts, but their efficiency for larger software development efforts which generally require more coordination and planning is questionable (Sommerville, 2004). Agile programming appears to be particularly suited for small to moderate size scientific computing software projects (Allen, 2009).

A popular form of the agile programming approach is extreme programming, or XP (Beck, 2000), and is so-called because it takes the standard software engineering practices to their extreme. In XP, requirements are expressed as potential scenarios that lead to software development tasks, which are then programmed by a pair of developers working as a team. Evidence suggests that pair programming productivity is similar to that of solo programmers (Williams and Kessler, 2003), but results in fewer errors since any code produced has necessarily undergone an informal software inspection process (Pressman, 2005). Unit tests (Section 4.3.3.1) must be developed for each task before the code is written, and all such tests must be successfully executed before integration into the software system, a process known as continuous integration testing (Duvall *et al.*, 2007). This type of test-first procedure also provides an implicit definition of the interface as well as proper usage examples of the component being developed. Software development projects employing XP usually have frequent releases and undergo frequent refactoring (Section 4.1.6) to improve quality and maintain simplicity. For an example of XP applied to scientific computing, see Wood and Kleb (2003).

4.1.5 Software reuse

Software reuse has become an important part of large software development projects (Sommerville, 2004). While its use in scientific computing is not as extensive, there are a number of areas in scientific computing where reuse is commonly found. Some examples of software reuse in scientific computing are: mathematical function and subroutine libraries (e.g., Press *et al.*, 2007), parallel message passing libraries (e.g., MPI), pre-packaged linear solvers such as the Linear Algebra Package (LAPACK) or the Portable Extensible Toolkit for Scientific Computation (PETSc), and graphics libraries.

4.1.6 Refactoring

Often, at the end of a software development effort, the developer realizes that choices made early in the software design phase have led to computationally inefficient or cumbersome programming. Refactoring is the act of modifying software such that the internal software structure is changed, but the outward behavior is not. Refactoring can reduce the complexity, computational time, and/or memory requirements for scientific software. However, refactoring should not be undertaken until a comprehensive test suite (Section 4.3.4) is in place to ensure that the external behavior is not modified and that programming errors are not introduced.

4.2 Version control

Version control tracks changes to source code or other software products. A good version control system can tell you what was changed, who made the change, and when the change was made. It allows a software developer to undo any changes to the code, going back to any prior version. This can be particularly helpful when you would like to reproduce results from an earlier paper, report, or project, and merely requires documentation of the version number or the date the results were generated. Version control also provides a mechanism for incorporating changes from multiple developers, an essential feature for large software projects or projects with geographically remote developers. All source code should be maintained in a version control system, regardless of how large or small the software project (Eddins, 2006).

Some key concepts pertaining to version control are discussed below (Collins-Sussman *et al.*, 2009). Note that the generic descriptor "file" is used, which could represent not only source code and other software products, but also any other type of file stored on a computer.

Repository	single location where the current and all prior versions of the files are stored. The repository can only be accessed through check-in and check-out procedures (see below).
Working copy	the local copy of a file from the repository which can be modified and then checked in to the repository.
Check-out	the process of creating a working copy from the repository, either from the current version or an earlier version.
Check-in	a check-in (or commit) occurs when changes made to a working copy are merged into the repository, resulting in a new version.
Diff	a summary of the differences between a working copy and a file in the repository, often taking the form of the two files shown side-by-side with differences highlighted.
Conflict	occurs when two or more developers attempt to make changes to the same file and the system is unable to reconcile the changes. Conflicts generally must be resolved by either choosing one version over the other or by integrating the changes from both into the repository by hand.
Update	merges recent changes to the repository from other developers into a working copy.
Version	a unique identifier assigned to each version of the file held in the repository which is generated by the check-in process.

The basic steps that one would use to get started with a version control tool are as follows. First, a *repository* is created, ideally on a network server which is backed up frequently. Then a software project (directory structure and/or files) is *imported* to the repository. This initial version can then be *checked-out* as a *working copy*. The project can then be modified in the working copy, with the differences between the edited working copy and the original repository version examined using a *diff* procedure. Before checking the working copy into the repository, two steps should be performed. First, an *update* should be performed to integrate changes that others have made in the code and to identify *conflicts*. Next, a set of predefined tests should be run to ensure that the modifications do not unexpectedly change the code's behavior. Finally, the *working copy* of the project can be *checked-in* to the repository, generating a new *version* of the software project.

There is a wide array of version control systems available to the software developer. These systems range from free, open-source systems such as Concurrent Versions Systems

(CVS) and Subversion (SVN) (Collins-Sussman *et al.*, 2009) to commercially available systems. A short tutorial showing the basic steps for implementing version control with a Windows-based tool can be found at www.aoe.vt.edu/∼cjroy/MISC/TortoiseSVN-Tutorial.pdf.

4.3 Software verification and validation

4.3.1 Definitions

The definitions accepted by AIAA (1998) and ASME (2006) for verification and validation as applied to scientific computing address the mathematical accuracy of a numerical solution (verification) and the physical accuracy of a given model (validation); however, the definitions used by the software engineering community (e.g., ISO, 1991; IEEE, 1991) are different. In software engineering, verification is defined as ensuring that software conforms to it specifications (i.e., requirements) and validation is defined as ensuring that software actually meets the customer's needs. Some argue that these definitions are really the same; however, upon closer examination, they are in fact different.

The key differences in these definitions for verification and validation are due to the fact that, in scientific computing, we begin with a governing partial differential or integral equation, which we will refer to as our mathematical model. For problems that we are interested in solving, there is generally no known exact solution to this model. It is for this reason that we must develop numerical approximations to the model (i.e., the numerical algorithm) and then implement that numerical algorithm within scientific computing software. Thus the two striking differences between how the scientific computing community and the software engineering community define verification and validation are as follows. First, in scientific computing, validation requires a comparison to experimental data. The software engineering community defines validation of the software as meeting the customer's needs, which is, in our opinion, too vague to tie it back to experimental observations. Second, in scientific computing, there is generally *no true system-level software test* (i.e., a test for correct code output given some code inputs) for real problems of interest. The "correct" output from the scientific software depends on the number of significant figures used in the computation, the computational mesh resolution and quality, the time step (for unsteady problems), and the level of iterative convergence. Chapters 5 and 6 of this book address the issue of system-level tests for scientific software.

In this section, we will distinguish between the two definitions of verification and validation by inserting the word "software" when referring to the definitions from software engineering. Three additional definitions that will be used throughout this section are those for software defects, faults, and failures (Hatton, 1997b). A *software defect* is a coding mistake (bug) or the misuse of a coding construct that could potentially lead to a software failure. A *software fault* is a defect which can be detected without running the code, i.e., through static analysis. Examples of defects that can lead to software faults include dependence on uninitialized variables, mismatches in parameter arguments, and unassigned pointers. A *software failure* occurs when the software returns an incorrect result or when it terminates prematurely due to a run-time error (overflow, underflow, division by zero, etc.). Some examples of catastrophic software failures are given by Hatton (1997a).

4.3.2 Static analysis

Static analysis is any type of assessment of software correctness that does not require program execution. Examples of static analysis methods include software inspection, peer review, compiling of the code, and the use of automatic static analyzers. Hatton (1997a) estimates that approximately 40% of software failures are due to static faults. Some examples of static faults are:

- dependency on uninitialized or undeclared variables,
- interface faults: too few, too many, or wrong type of arguments passed to a function/subroutine,
- casting a pointer to a narrower integer type (C), and
- use of non-local variables in functions/subroutines (Fortran).

All of these static faults, as well as others that have their origins in ambiguities in the programming language standards, can be prevented by using static analysis.

4.3.2.1 Software inspection

Software inspection (or review) refers to the act of reading through the source code and other software products to find defects. Although software inspections are time intensive, they are surprisingly effective at finding software defects (Sommerville, 2004). Other advantages of software inspections are that they are not subject to interactions between different software defects (i.e., one defect will not hide the presence of another one), incomplete and nonfunctional source code can be inspected, and they can find other issues besides defects such as coding inefficiencies or lack of compliance with coding standards. The rigor of the software inspection depends on the technical qualifications of the reviewer as well as their level of independence from the software developers.

4.3.2.2 Compiling the code

Any time the code is compiled it goes through some level of static analysis. The level of rigor of the static analysis often depends on the options used during compilation, but there is a trade-off between the level of static analysis performed by the compiler and the execution speed. Many modern compilers provide different modes of compilation such as a release mode, a debug mode, and a check mode that perform increasing levels of static analysis. Due to differences in compilers and operating systems, many software developers make it standard practice to compile the source code with different compilers and on different platforms.

4.3.2.3 Automatic static analyzers

Automatic static analyzers are external tools that are meant to complement the checking of the code by the compiler. They are designed to find inconsistent or undefined use of a programming language that the compiler will likely overlook, as well as coding constructs that are generally considered as unsafe. Some static analyzers available for C/C^{++} include the Safer C Toolset, CodeWizard, CMT^{++}, Cleanscape LintPlus, PC-lint/FlexeLint, and QA C. Static analyzers for Fortran include floppy/fflow and ftnchek. There is also a recently-developed static analyzer for MATLAB called M-Lint (MATLAB, 2008). For a more complete list, or for references to each of these static analyzers, see www.testingfaqs.org/t-static.html.

4.3.3 Dynamic testing

Dynamic software testing can be defined as the "dynamic verification of the behavior of a program on a finite set of test cases . . . against the expected behavior" (SWEBOK, 2004). Dynamic testing includes any type of testing activity which involves running the code, thus run-time compiler checks (e.g., array bounds checking, pointer checking) fall under the heading of dynamic testing. The types of dynamic testing discussed in this section include defect testing (at the unit, component, and complete system level), regression testing, and software validation testing.

4.3.3.1 Defect testing

Defect testing is a type of dynamic testing performed to uncover the presence of a software defect; however, defect testing cannot be used to prove that no errors are present. Once a defect is discovered, the process of finding and fixing the defect is usually referred to as debugging. In scientific computing, it is convenient to decompose defect testing into three levels: unit testing which occurs at the smallest level in the code, component testing which occurs at the submodel or algorithm level, and system testing where the desired output from the software is evaluated. While unit testing is generally performed by the code developer, component and system-level testing is more reliable when performed by someone outside of the software development team.

Unit testing

Unit testing is used to verify the execution of a single routine (e.g., function, subroutine, object class) of the code (Eddins, 2006). Unit tests are designed to check for the correctness of routine output based on a given input. They should also be easy to write and run, and should execute quickly. Properly designed unit tests also provide examples of proper routine use such as how the routine should be called, what type of inputs should be provided, what type of outputs can be expected.

While it does take additional time to develop unit tests, this extra time in code development generally pays off later in reduced time debugging. The authors' experience with even

Table 4.1 *Example of a component-level test fixture for Sutherland's viscosity law (adapted from Kleb and Wood, 2006).*

Input: T (K)	Output: μ (kg/s-m)
$200 \leq T \leq 3000$	$B^* \frac{T^{1.5}}{T+110.4}$
199	error
200	1.3285589×10^{-5}
2000	6.1792781×10^{-5}
3000	7.7023485×10^{-5}
3001	error

* where $B = 1.458 \times 10^{-6}$

small scientific computing code development in university settings suggests that the typical ratio of debugging time to programming time for students who do not employ unit tests is at least five to one. The wider the unit testing coverage (i.e., percentage of routines that have unit tests), the more reliable the code is likely to be. In fact, some software development strategies such as Extreme Programming (XP) require tests to be written before the actual routine to be tested is created. Such strategies require the programmer to clearly define the interfaces (inputs and outputs) of the routine up front.

Component testing

Kleb and Wood (2006) make an appeal to the scientific computing community to implement the scientific method in the development of scientific software. Recall that, in the scientific method, a theory must be supported with a corresponding experiment that tests the theory, and must be described in enough detail that the experiment can be reproduced by independent sources. For application to scientific computing, they recommend testing at the component level, where a component is considered to be a submodel or algorithm. Furthermore, they strongly suggest that model and algorithm developers publish *test fixtures* with any newly proposed model or algorithm. These test fixtures are designed to clearly define the proper usage of the component, give examples of proper usage, and give sample inputs along with correct outputs that can be used for testing the implementation in a scientific computing code. An example of such a test fixture for Sutherland's viscosity law is presented in Table 4.1.

Component-level testing can be performed when the submodel or algorithm are algebraic since the expected (i.e., correct) solution can be computed directly. However, for cases where the submodel involves numerical approximations (e.g., many models for fluid turbulence involve differential equations), then the expected solution will necessarily be a function of the chosen discretization parameters, and the more sophisticated code verification methods discussed in Chapters 5 and 6 should be used. For models that are difficult to test at the system level (e.g., the `min` and `max` functions significantly complicate the code verification process discussed in Chapter 5), component-level testing of the models (or different parts of the model) can be used. Finally, even when all components have been successfully tested individually, one should not get a false sense of security about how the software will behave at the system level. Complex interactions between components can only be tested at the system level.

System testing

System-level testing addresses code as a whole. For a given set of inputs to the code, what is the correct code output? In software engineering, system level testing is the primary means by which one determines if the software requirements have been met (i.e., software verification). For nonscientific software, it is often possible to *a priori* determine what the correct output of the code should be. However, for scientific computing software where partial differential or integral equations are solved, the “correct” output is generally not known ahead of time. Furthermore, the code output will depend on the grid and time step chosen, the iterative convergence level, the machine precision, etc. For scientific computing software, system-level testing is generally addressed through *order of accuracy verification*, which is the main subject of Chapter 5.

4.3.3.2 Regression testing

Regression testing involves the comparison of code or software routine output to the output from earlier versions of the code. Regression tests are designed to prevent the introduction of coding mistakes by detecting unintended consequences of changes in the code. Regression tests can be implemented at the unit, component, and system level. In fact, all of the defect tests described above can also be implemented as regression tests. The main difference between regression testing and defect testing is that regression tests do not compare code output to the correct expected value, but instead to the output from previous versions of the code. Careful regression testing combined with defect testing can minimize the chances of introducing new software defects during code development and maintenance.

4.3.3.3 Software validation testing

As discussed earlier, software validation is performed to ensure that the software actually meets the customer’s needs in terms of software function, behavior, and performance (Pressman, 2005). Software validation (or acceptance) testing occurs at the system level and usually involves data supplied by the customer. Software validation testing for scientific computing software inherits all of the issues discussed earlier for system-level testing, and thus special considerations must be made when determining what the expected, correct output of the code should be.

4.3.4 Test harness and test suites

Many different types of dynamic software tests have been discussed in this section. For larger software development projects, it would be extremely tedious if the developer had to run each of the tests separately and then examine the results. Especially in the case of larger development efforts, automation of software testing is a must.

A *test harness* is the combination of software and test data used to test the correctness of a program or component by automatically running it under various conditions (Eddins, 2006). A test harness is usually composed of a test manager, test input data, test output data, a file comparator, and an automatic report generator. While it is certainly possible to create your own test harness, there are a variety of test harnesses that have been developed for a wide range of programming languages. For a detailed list, see en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

Once a suite of tests has been set up to run within a test harness, it is recommended that these tests be run automatically at specified intervals. Shorter tests can be run in a nightly test suite, while larger tests which require more computer time and memory may be set up in weekly or monthly test suites. In addition, an approach called *continuous integration testing* (Duvall *et al.*, 2007) requires that specified test suites be run before any new code modifications are checked in.

4.3.5 Code coverage

Regardless of how software testing is done, one important aspect is the coverage of the tests. *Code coverage* can be defined as the percentage of code components (and possibly their interactions) for which tests exist. While testing at the unit and component levels is relatively straightforward, system-level testing must also address interactions between different components. Large, complex scientific computing codes generally have a very large number of options for models, submodels, numerical algorithms, boundary conditions, etc. Assume for the moment that there are 100 different options in the code to be tested, a conservative estimate for most production-level scientific computing codes. Testing each option independently (although generally not possible) would require 100 different system-level tests. Testing pair-wise combinations for interactions between these different options would require 4950 system level tests. Testing the interactions between groups of three would require 161 700 tests. While this is clearly an upper bound since many options may be mutually exclusive, it does provide a sense of the magnitude of the task of achieving complete code coverage of model/algorithm interactions. Table 4.2 provides a comparison of the number of system-level tests required to ensure code coverage with different degrees of option interactions for codes with 10, 100, and 1000 different code options. Clearly, testing the three-way interactions for our example of 100 coding options is impossible, as would be testing all pair-wise interactions when 1000 coding options are available, a number not uncommon for commercial scientific computing codes. One possible approach for addressing this combinatorial explosion of tests for component interactions is

Table 4.2 *Number of system-level tests required for complete code coverage for codes with different numbers of options and option combinations to be tested.*

Number of options	Option combinations to be tested	System-level tests required
10	1	10
10	2	45
10	3	720
100	1	100
100	2	4950
100	3	161 700
1000	1	1000
1000	2	499 500
1000	3	$\sim 1.7 \times 10^8$

application-centric testing (Knupp and Ober, 2008), where only those components and component interactions which impact a specific code application are tested.

4.3.6 Formal methods

Formal methods use mathematically-based techniques for requirements specification, development, and/or verification testing of software systems. Formal methods arise from discrete mathematics and involve set theory, logic, and algebra (Sommerville, 2004). Such a rigorous mathematical framework is expensive to implement, thus it is mainly used for high-assurance (i.e., critical) software systems such as those found in aircraft controls systems, nuclear power plants, and medical devices such as pacemakers (Heitmeyer, 2004). Some of the drawbacks to using formal methods are that they do not handle user interfaces well and they do not scale well for larger software projects. Due to the effort and expense required, as well as their poor scalability, we do not recommend formal methods for scientific computing software.

4.4 Software quality and reliability

There are many different definitions of quality applied to software. The definition that we will use is: *conformance to customer requirements and needs*. This definition implies not only adherence to the formally documented requirements for the software, but also those requirements that are not explicitly stated by the customer that need to be met. However, this definition of quality can often only be applied after the complete software product is delivered to the customer. Another aspect of software quality that we will find useful is software reliability. One definition of *software reliability* is the probability of failure-free operation of software in a given environment for a specified time (Musa, 1999). In this section we present some explicit and implicit methods for measuring software reliability. A discussion of recommended programming practices as well as error-prone coding constructs that should be avoided when possible, both of which can affect software reliability, can be found in the Appendix.

4.4.1 Reliability metrics

Two quantitative approaches for measuring code quality are *defect density analysis*, which provides an explicit measure of reliability, and *complexity analysis*, which provides an implicit measure of reliability. Additional information on software reliability can be found in Beizer (1990), Fenton and Pfleeger (1997), and Kaner *et al.* (1999).

4.4.1.1 Defect density analysis

The most direct method for assessing the reliability of software is in terms of the number of defects in the software. Defects can lead to static errors (faults) and dynamic errors (failures). The *defect density* is usually reported as the number of defects per executable source lines of code (SLOC). Hatton (1997a) argues that it is only by measuring the defect density of software, through both static analysis and dynamic testing, that an objective assessment of software reliability can be made. Hatton's T Experiments (Hatton, 1997b) are discussed in detail in the next section and represent the largest known defect density study of scientific software.

A significant limitation of defect density analysis is that the defect rate is a function of both the number of defects in the software and the specific testing procedure used to find the defects (Fenton and Pfleeger, 1997). For example, a poor testing procedure might uncover only a few defects, whereas a more comprehensive testing procedure applied to the same software might uncover significantly more defects. This sensitivity to the specific approach used for defect testing represents a major limitation of defect density analysis.

4.4.1.2 Complexity analysis

Complexity analysis is an indirect way of measuring reliability because it requires a model to convert internal code quality attributes into code reliability (Sommerville, 2004). The most frequently used model is to assume that a high degree of complexity in a component (function, subroutine, object class, etc.) is bad while a low degree of complexity is good. In this case, components which are identified as being too complex can be decomposed into smaller components. However, Hatton (1997a) used defect density analysis to show that the defect density in components follows a U-shaped curve, with the minimum occurring at 150–250 lines of source code per component, independent of both programming language and application area. He surmised that the increase in defect density for smaller components may be related to the inadvertent adverse effects of component reuse (see Hatton (1996) for more details). Some different internal code attributes that can be used to indirectly assess code reliability are discussed in this subsection, and in some cases, tools exist for automatically evaluating these complexity metrics.

Lines of source code

The simplest measure of complexity can be found by counting the number of executable source lines of code (SLOC) for each component. Hatton (1997a) recommends keeping components between 150 and 250 SLOC.

NPATH metric

The NPATH metric simply counts the number of possible execution paths through a component (Nejmeh, 1988). Nejmeh (1988) recommends keeping this value below 200 for each component.

Cyclomatic complexity

The cyclomatic, or McCabe, complexity (McCabe, 1976) is defined as one plus the number of decision points in a component, where a decision point is defined as any loop or logical statement (`if, elseif, while, repeat, do, for, or`, etc.). The maximum recommended value for cyclomatic complexity of a component is ten (Eddins, 2006).

Depth of conditional nesting

This complexity metric provides a measure of the depth of nesting of if-statements, where larger degrees of nesting are assumed to be more difficult to understand and track, and therefore are more error prone (Sommerville, 2004).

Depth of inheritance tree

Applicable to object-oriented programming languages, this complexity metric measures the number of levels in the inheritance tree where sub-classes inherit attributes from super-classes (Sommerville, 2004). The more levels that exist in the inheritance tree, the more classes one needs to understand to be able to develop or modify a given object class.

4.5 Case study in reliability: the T experiments

In the early 1990s, Les Hatton undertook a broad study of scientific software reliability known collectively as the “T Experiments” (Hatton, 1997b). This study was broken into two parts: the first (T1) examined codes from a wide range of scientific disciplines using static analysis, while the second (T2) examined codes in a single discipline using dynamic testing.

The T1 study used static deep-flow analyzers to examine more than 100 different codes in 40 different application areas. All codes were written in C, FORTRAN 66, or FORTRAN 77, and the static analyzers used were QA C (for the C codes) and QA Fortran (for the

FORTRAN codes). The main conclusion of the T1 study was that the C codes contained approximately eight serious static faults per 1000 lines of executable code, while the FORTRAN codes contained approximately 12 faults per 1000 lines. A serious static fault is defined as a statically-detectable defect that is likely to cause the software to fail. For more details on the T1 study, see Hatton (1995).

The T2 study examined a subset of the codes from the T1 study in the area of seismic data processing which is used in the field of oil and gas exploration. This study examined nine independent, mature, commercial codes which employed the same algorithms, the same programming language (FORTRAN), the same user-defined parameters, and the same input data. Hatton refers to such a study as N-version programming since each code was developed independently by a different company. Each of the codes consisted of approximately 30 sequential steps, 14 of which used unambiguously defined algorithms, referred to in the study as primary calibration points. Agreement between the codes after the first primary calibration point was within 0.001% (i.e., approximately machine precision for single-precision computations); however, agreement after primary calibration point 14 was only within a factor of two. It is interesting to note that distribution of results from the various codes was found to be non-Gaussian with distinct groups and outliers, suggesting that the output from an N-version programming test should not be analyzed with Bayesian statistics. Hatton concluded that the disagreements between the different codes are due primarily to software errors. Such dismal results from the T2 study prompted Hatton to conclude that *"the results of scientific calculations carried out by many software packages should be treated with the same measure of disbelief researchers have traditionally attached to the results of unconfirmed physical experiments."* For more details on the T2 study, see Hatton and Roberts (1994).

These alarming results from Hatton's "T Experiments" highlight the need for employing good software engineering practices in scientific computing. At a minimum, the simple techniques presented in this chapter such as version control, static analysis, dynamic testing, and reliability metrics should be employed for all scientific computing software projects to improve quality and reliability.

4.6 Software engineering for large software projects

Up to this point, the software engineering practices discussed have been applicable to all scientific computing project whether large or small. In this section, we specifically address software engineering practices for large scientific computing projects that may be less effective for smaller projects. The two broad topics addressed here include software requirements and software management.

4.6.1 Software requirements

A software requirement is a "property that must be exhibited in order to solve some real-world problem" (SWEBOK, 2004). Uncertainty in requirements is a leading cause of failure in software projects (Post and Kendall, 2004). While it is certainly ideal to have all requirements rigorously and unambiguously specified at the beginning of a software project, this can be difficult to achieve for scientific software. Especially in the case of large scientific software development projects, complete requirements can be difficult to specify due to rapid changes in models, algorithms, and even in the specialized computer architectures used to run the software. While lack of requirements definition can adversely affect the development of scientific software, these negative effects can be mitigated somewhat if close communication is maintained between the developer of the software and the user (Post and Kendall, 2004) or if the developer is also an expert in the scientific computing discipline.

4.6.1.1 Types of software requirements

There are two main types of software requirements. User requirements are formulated at a high level of abstraction, usually in general terms which are easily understood by the user. An example of a user requirement might be: *this software should produce approximate numerical solutions to the Navier–Stokes equations*. Software system requirements, on the other hand, are a precise and formal definition of a software system's functions and constraints. The software system requirements are further decomposed as follows:

1 functional requirements – rigorous specifications of required outputs for a given set of inputs,
2 nonfunctional requirements – additional nonfunctional constraints such as programming standards, reliability, and computational speed, and
3 domain requirements – those requirements that come from the application domain such as a discussion of the partial differential or integral equations to be solved numerically for a given scientific computing application.

The domain requirements are crucial in scientific computing since these will be used to define the specific governing equations, models, and numerical algorithms to be implemented. Finally, if the software is to be integrated with existing software, then additional specifications may be needed for the procedure interfaces (application programming interfaces, or APIs), data structures, or data representation (e.g., bit ordering) (Sommerville, 2004).

4.6.1.2 Requirements engineering process

The process for determining software requirements contains four phases: elicitation, analysis, specification, and validation. *Elicitation* involves the identification of the sources for requirements, which includes the code customers, users, and developers. For larger software projects these sources could also include managers, regulatory authorities, third-party software providers, and other stakeholders. Once the sources for the requirements have been identified, the requirements are then collected either individually from those sources or by bringing the sources together for discussion.

In the *analysis* phase, the requirements are analyzed for clarity, conflicts, and the need for possible requirements negotiation between the software users and developers. In scientific computing, while the users typically want a code with a very broad range of capabilities, the developers must weigh trade-offs between capability and the required computational infrastructure, all while operating under manpower and budgetary constraints. Thus negotiation and compromise between the users and the developers is critical for developing computational tools that balance capability with feasibility and available resources.

Specification deals with the documentation of the established user and system requirements in a formal software requirements document. This requirements document should be considered a living document since requirements often change during the software's life cycle. Requirements *validation* is the final confirmation that the software meets the customer's needs, and typically comes in the form of full software system tests using data supplied by the customer. One challenge specific to scientific computing software is the difficulty in determining the correct code output due to the presence of numerical approximation errors.

4.6.1.3 Requirements management

Requirements management is the process of understanding, controlling, and tracking changes to the system requirements. It is important because software requirements are usually incomplete and tend to undergo frequent changes. Things that can cause the requirements to change include installing the software on a new hardware system, identification of new desired functionality based on user experience with the software, and, for scientific computing, improvements in existing models or numerical algorithms.

4.6.2 Software management

Software management is a broad topic which includes the management of the software project, cost, configuration, and quality. In addition, effective software management strategies must include approaches for improvement of the software development process itself.

4.6.2.1 Project management

Software project management addresses the planning, scheduling, oversight, and risk management of a software project. For larger projects, planning activities encompass a wide range of different areas, and separate planning documents should be developed for quality, software verification and validation, configuration management, maintenance, staff development, milestones, and deliverables. Another important aspect of software project management is determining the level of formality required in applying the software engineering practices. Ultimately, this decision should be made by performing a risk-based assessment of the intended use, mission, complexity, budget, and schedule (Demarco and Lister, 2003).

Managing software projects is generally more difficult than managing standard engineering projects because the product is intangible, there are usually no standard software management practices, and large software projects are usually one-of-a-kind endeavors

(Sommerville, 2004). According to Post and Kendall (2004), ensuring consistency between the software schedule, resources, and requirements is the key to successfully managing a large scientific computing software project.

4.6.2.2 Cost estimation

While estimating the required resources for a software project can be challenging, semi-empirical models are available. These models are called algorithmic cost models, and in their simplest form (Sommerville, 2004) can be expressed as:

$$Effort = A \times (Size)^b \times M. \tag{4.1}$$

In this simple algorithmic cost model, A is a constant which depends on the type of organization developing the software, their software development practices, and the specific type of software being developed. *Size* is some measure of the size of the software project (estimated lines of code, software functionality, etc.). The exponent b typically varies between 1 and 1.5, with larger values indicative of the fact that software complexity increases nonlinearly with the size of the project. M is a multiplier that accounts for various factors including risks associated with software failures, experience of the code development team, and the dependability of the requirements. *Effort* is generally in man-months, and the cost is usually assumed to be proportional to the effort. Most of these parameters are subjective and difficult to evaluate, thus they should be determined empirically using historical data for the organization developing the software whenever possible. When such data are not available, historical data from similar organizations may be used.

For larger software projects where more accurate cost estimates are required, Sommerville (2004) recommends the more detailed Constructive Cost Model (COCOMO). When software is developed using imperative programming languages such as Fortran or C using a waterfall model, the original COCOMO model, now referred to as COCOMO 81, can be used (Boehm, 1981). This algorithmic cost model was developed by Boehm while he was at the aerospace firm TRW Inc., and drew upon the historical software development data from 63 different software projects ranging from 2000 to 10000 lines of code. An updated model, COCOMO II, has been developed which accounts for object-oriented programming languages, software reuse, off-the-shelf software components, and a spiral software development model (Boehm *et al.*, 2000).

4.6.2.3 Configuration management

Configuration management deals with the control and management of the software products during all phases of the software product's lifecycle including planning, development, production, maintenance, and retirement. Here software products include not only the source code, but also user and theory manuals, software tests, test results, design documents, web pages, and any other items produced during the software development process. Configuration management tracks the way software is configured over time and is used for controlling changes to, and for maintaining integrity and traceability of, the software products

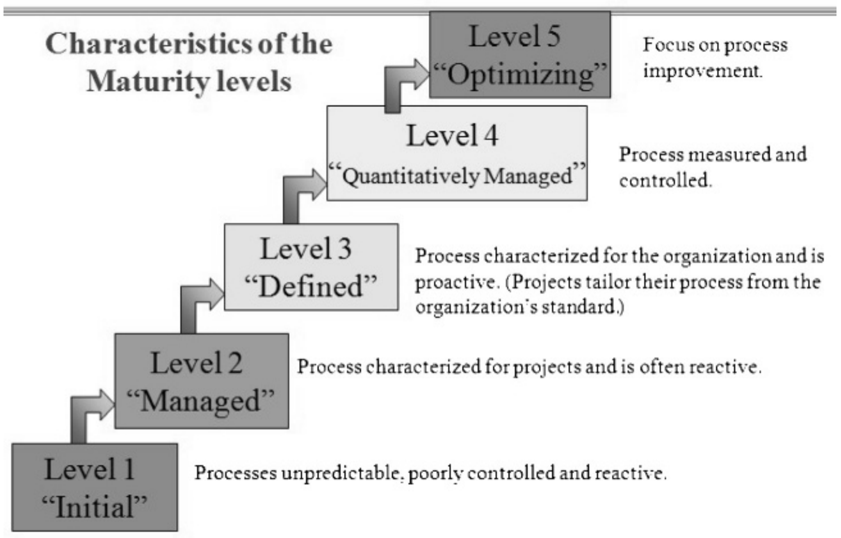
Characteristics of the Maturity levels
Level 5 "Optimizing"
Focus on process improvement.
Level 4 "Quantitatively Managed"
Process measured and controlled.
Level 3 "Defined"
Process characterized for the organization and is proactive. (Projects tailor their process from the organization's standard.)
Level 2 "Managed"
Process characterized for projects and is often reactive.
Level 1 "Initial"
Processes unpredictable, poorly controlled and reactive.

Figure 4.2 Characteristics of the maturity levels in CMMI (from Godfrey, 2009).

(Sommerville, 2004). The key aspects of configuration management include using *version control* (discussed in Section 4.2) for source code and other important software products, identification of the software products to be managed, recording, approving, and tracking issues with the software, managing software releases, and ensuring frequent backups are made.

4.6.2.4 Quality management

Software quality management is usually separated into three parts: quality assurance, quality planning, and quality control (Sommerville, 2004). *Quality assurance* is the definition of a set of procedures and standards for developing high-quality software. *Quality planning* is the process of selecting from the above procedures and standards for a given software project. *Quality control* is a set of processes that ensure the quality plan was actually implemented. It is important to maintain independence between the quality management team and the code development team (Sommerville, 2004).

4.6.2.5 Process improvement

Another way to improve the quality of software is to improve the processes which are used to develop it. Perhaps the most well-known software process improvement model is the Capability Maturity Model, or CMM (Humphrey, 1989). The successor to CMM, the Capability Maturity Model Integration (CMMI) integrates various process improvement models and is more broadly applicable to the related areas of systems engineering and integrated product development (SEI, 2009). The five maturity levels in CMMI are shown in Figure 4.2, and empirical evidence suggests that both software quality and developer productivity will improve as higher levels of process maturity are reached (Gibson *et al.*, 2006).

Post and Kendall (2004) found that not all software engineering practices are helpful for developing scientific software. They cautioned against blindly applying rigorous software standards such as CMM/CMMI without first performing a cost-benefit analysis. Neely (2004) suggests a risk-based approach to applying quality assurance practices to scientific computing projects. High-risk projects are defined as those that could potentially involve "great loss of money, reputation, or human life," while a low risk project would involve at most inconvenience to the user. High-risk projects would be expected to conform to more formal software quality standards, whereas low-risk projects would allow more informal, ad-hoc implementation of the standards.

4.7 References

AIAA (1998). *Guide for the Verification and Validation of Computational Fluid Dynamics Simulations*. AIAA-G-077–1998, Reston, VA, American Institute of Aeronautics and Astronautics.

Allen, E. B. (2009). Private communication, February 11, 2009.

ASME (2006). *Guide for Verification and Validation in Computational Solid Mechanics*. ASME V&V 10–2006, New York, NY, American Society of Mechanical Engineers.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*, Reading, PA, Addison-Wesley.

Beizer, B. (1990). *Software Testing Techniques*, 2nd edn., New York, Van Nostrand Reinhold.

Boehm, B. W. (1981). *Software Engineering Economics*, Englewood Cliffs, NJ, Prentice-Hall.

Boehm, B. W., C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece (2000). *Software Cost Estimation with Cocomo II*, Englewood Cliffs, NJ, Prentice-Hall.

Collins-Sussman, B., B. W. Fitzpatrick, and C. M. Pilato (2009). *Version Control with Subversion: For Subversion 1.5: (Compiled from r3305)* (see svnbook.red-bean.com/en/1.5/svn-book.pdf).

Demarco, T. and T. Lister (2003). *Waltzing with Bears: Managing Risk on Software Projects*, New York, Dorset House.

Duvall, P. F., S. M. Matyas, and A. Glover (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*, Upper Saddle River, NJ, Harlow: Addison-Wesley.

Eddins, S. (2006). Taking control of your code: essential software development tools for engineers, *International Conference on Image Processing*, Atlanta, GA, Oct. 9 (see blogs.mathworks.com/images/steve/92/handout_final_icip2006.pdf).

Fenton, N. E. and S. L. Pfleeger (1997). *Software Metrics: a Rigorous and Practical Approach*, 2nd edn., London, PWS Publishing.

Gibson, D. L., D. R. Goldenson, and K. Kost (2006). *Performance Results of CMMI®-Based Process Improvement*, Technical Report CMU/SEI-2006-TR-004, ESC-TR-2006–004, August 2006 (see www.sei.cmu.edu/publications/documents/06.reports/06tr004.html).

Godfrey, S. (2009). *What is CMMI?* NASA Presentation (see software.gsfc.nasa.gov/docs/What%20is%20CMMI.ppt).

Hatton, L. (1995). *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*, New York, McGraw-Hill International Ltd.

Hatton, L. (1996). Software faults: the avoidable and the unavoidable: lessons from real systems, *Proceedings of the Product Assurance Workshop, ESA SP-377*, Noordwijk, The Netherlands.

Hatton, L. (1997a). Software failures: follies and fallacies, *IEEE Review*, March, 49–52.

Hatton, L. (1997b). The T Experiments: errors in scientific software, *IEEE Computational Science and Engineering*, **4**(2), 27–38.

Hatton, L., and A. Roberts (1994). How accurate is scientific software? *IEEE Transactions on Software Engineering*, **20**(10), 785–797.

Heitmeyer, C. (2004). Managing complexity in software development with formally based tools, *Electronic Notes in Theoretical Computer Science*, **108**, 11–19.

Humphrey, W. (1989). *Managing the Software Process*. Reading, MA, Addison-Wesley Professional.

IEEE (1991). *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12–1990, New York, IEEE.

ISO (1991). *ISO 9000–3: Quality Management and Quality Assurance Standards – Part 3: Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software*. Geneva, Switzerland, International Organization for Standardization.

Kaner, C., J. Falk, and H. Q. Nguyen (1999). *Testing Computer Software*, 2nd edn., New York, Wiley.

Kleb, B., and B. Wood (2006). Computational simulations and the scientific method, *Journal of Aerospace Computing, Information, and Communication*, **3**(6), 244–250.

Knupp, P. M. and C. C. Ober (2008). *A Code-Verification Evidence-Generation Process Model and Checklist*, Sandia National Laboratories Report SAND2008–4832.

Knupp, P. M., C. C., Ober, and R. B. Bond (2007). *Impact of Coding Mistakes on Numerical Error and Uncertainty in Solutions to PDEs*, Sandia National Laboratories Report SAND2007–5341.

MATLAB (2008). *MATLAB® Desktop Tools and Development Environment*, Natick, MA, The Mathworks, Inc. (see www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matlab_env.pdf).

McCabe, T. J. (1976). A complexity measure, *IEEE Transactions on Software Engineering*, **2**(4), 308–320.

McConnell, S. (2004). *Code Complete: a Practical Handbook of Software Construction*, 2nd edn., Redmond, WA, Microsoft Press.

Musa, J. D. (1999). *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, New York, McGraw-Hill.

Neely, R. (2004). Practical software quality engineering on a large multi-disciplinary HPC development team, *Proceedings of the First International Workshop on Software Engineering for High Performance Computing System Applications*, Edinburgh, Scotland, May 24, 2004.

Nejmeh, B. A. (1988). Npath: a measure of execution path complexity and its applications, *Communications of the Association for Computing Machinery*, **31**(2), 188–200.

Post, D. E., and R. P. Kendall (2004). Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: lessons learned from ASCI, *International Journal of High Performance Computing Applications*, **18**(4), 399–416.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (2007). *Numerical Recipes: the Art of Scientific Computing*, 3rd edn., Cambridge, Cambridge University Press.

Pressman, R. S. (2005). *Software Engineering: a Practitioner's Approach*, 6th edn., Boston, MA, McGraw-Hill.

Roy, C. J. (2009). Practical software engineering strategies for scientific computing, AIAA Paper 2009–3997, *19th AIAA Computational Fluid Dynamics, San Antonio, TX*, June 22–25, 2009.

SE-CSE (2008). *Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering*, Leipzig, Germany, May 13, 2008 (see cs.ua.edu/∼SECSE08/).

SE-CSE (2009). *Proceedings of the Second International Workshop on Software Engineering for Computational Science and Engineering*, Vancouver, Canada, May 23, 2009 (see cs.ua.edu/∼SECSE09/).

SE-HPC (2004). *Proceedings of the First International Workshop On Software Engineering for High Performance Computing System Applications*, Edinburgh, Scotland, May 24, 2004.

SEI (2009). CMMI Main Page, Software Engineering Institute, Carnegie Mellon University (see www.sei.cmu.edu/cmmi/index.html).

Sommerville, I. (2004). *Software Engineering*, 7th edn., Harlow, Essex, England, Pearson Education Ltd.

SWEBOK (2004), *Guide to the Software Engineering Body of Knowledge: 2004 Edition*, P. Borque and R. Dupuis (eds.), Los Alamitos, CA, IEEE Computer Society (www.swebok.org).

Williams, L. and R. Kessler (2003). *Pair Programming Illuminated*, Boston, MA, Addison-Wesley.

Wilson, G. (2009). *Software Carpentry*, www.swc.scipy.org/.

Wood, W. A. and W. L. Kleb (2003). Exploring XP for scientific research, *IEEE Software*, **20**(3), 30–36.