10-20-2016

Software Engineering for Science

Jeffrey C. Carver
University of Alabama - Tuscaloosa

Neil P. Chue Hong
University of Edinburgh

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Recommended Citation

Jeffrey C. Carver, Neil Chue P. Hong, and George K. Thiruvathukal (editors), Software Engineering for Science, Taylor and Francis/CRC Press.

SOFTWARE ENGINEERING FOR SCIENCE

Chapman & Hall/CRC Computational Science Series

SERIES EDITOR

Horst Simon
Deputy Director
Lawrence Berkeley National Laboratory
Berkeley, California, U.S.A.

PUBLISHED TITLES

COMBINATORIAL SCIENTIFIC COMPUTING
Edited by Uwe Naumann and Olaf Schenk

CONTEMPORARY HIGH PERFORMANCE COMPUTING: FROM PETASCALE TOWARD EXASCALE
Edited by Jeffrey S. Vetter

CONTEMPORARY HIGH PERFORMANCE COMPUTING: FROM PETASCALE TOWARD EXASCALE, VOLUME TWO
Edited by Jeffrey S. Vetter

DATA-INTENSIVE SCIENCE
Edited by Terence Critchlow and Kerstin Kleese van Dam

THE END OF ERROR: UNUM COMPUTING
John L. Gustafson

FROM ACTION SYSTEMS TO DISTRIBUTED SYSTEMS: THE REFINEMENT APPROACH
Edited by Luigia Petre and Emil Sekerinski

FUNDAMENTALS OF MULTICORE SOFTWARE DEVELOPMENT
Edited by Victor Pankratius, Ali-Reza Adl-Tabatabai, and Walter Tichy

FUNDAMENTALS OF PARALLEL MULTICORE ARCHITECTURE
Yan Solihin

THE GREEN COMPUTING BOOK: TACKLING ENERGY EFFICIENCY AT LARGE SCALE
Edited by Wu-chun Feng

GRID COMPUTING: TECHNIQUES AND APPLICATIONS
Barry Wilkinson

HIGH PERFORMANCE COMPUTING: PROGRAMMING AND APPLICATIONS
John Levesque with Gene Wagenbreth

HIGH PERFORMANCE PARALLEL I/O
Prabhat and Quincey Koziol

PUBLISHED TITLES CONTINUED

HIGH PERFORMANCE VISUALIZATION:
ENABLING EXTREME-SCALE SCIENTIFIC INSIGHT
Edited by E. Wes Bethel, Hank Childs, and Charles Hansen

INDUSTRIAL APPLICATIONS OF HIGH-PERFORMANCE COMPUTING:
BEST GLOBAL PRACTICES
Edited by Anwar Osseyran and Merle Giles

INTRODUCTION TO COMPUTATIONAL MODELING USING C AND
OPEN-SOURCE TOOLS
José M Garrido

INTRODUCTION TO CONCURRENCY IN PROGRAMMING LANGUAGES
Matthew J. Sottile, Timothy G. Mattson, and Craig E Rasmussen

INTRODUCTION TO ELEMENTARY COMPUTATIONAL MODELING: ESSENTIAL
CONCEPTS, PRINCIPLES, AND PROBLEM SOLVING
José M. Garrido

INTRODUCTION TO HIGH PERFORMANCE COMPUTING FOR SCIENTISTS
AND ENGINEERS
Georg Hager and Gerhard Wellein

INTRODUCTION TO REVERSIBLE COMPUTING
Kalyan S. Perumalla

INTRODUCTION TO SCHEDULING
Yves Robert and Frédéric Vivien

INTRODUCTION TO THE SIMULATION OF DYNAMICS USING SIMULINK®
Michael A. Gray

PEER-TO-PEER COMPUTING: APPLICATIONS, ARCHITECTURE, PROTOCOLS,
AND CHALLENGES
Yu-Kwong Ricky Kwok

PERFORMANCE TUNING OF SCIENTIFIC APPLICATIONS
Edited by David Bailey, Robert Lucas, and Samuel Williams

PETASCALE COMPUTING: ALGORITHMS AND APPLICATIONS
Edited by David A. Bader

PROCESS ALGEBRA FOR PARALLEL AND DISTRIBUTED PROCESSING
Edited by Michael Alexander and William Gardner

SCIENTIFIC DATA MANAGEMENT: CHALLENGES, TECHNOLOGY, AND DEPLOYMENT
Edited by Arie Shoshani and Doron Rotem

SOFTWARE ENGINEERING FOR SCIENCE
Edited by Jeffrey C. Carver, Neil P. Chue Hong, and George K. Thiruvathukal

SOFTWARE ENGINEERING FOR SCIENCE

Edited by

Jeffrey C. Carver
University of Alabama, USA

Neil P. Chue Hong
University of Edinburgh, UK

George K. Thiruvathukal
Loyola University Chicago, Chicago, Illinois

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20160817

International Standard Book Number-13: 978-1-4987-4385-3 (Hardback)

Library of Congress Cataloging-in-Publication Data

Names: Carver, Jeffrey, editor. | Hong, Neil P. Chue, editor. |
Thiruvathukal, George K. (George Kuriakose), editor.
Title: Software engineering for science / edited by Jeffrey Carver, Neil P.
Chue Hong, and George K. Thiruvathukal.
Description: Boca Raton : Taylor & Francis, CRC Press, 2017. | Series:
Computational science series ; 30 | Includes bibliographical references
and index.
Identifiers: LCCN 2016022277 | ISBN 9781498743853 (alk. paper)
Subjects: LCSH: Science--Data processing. | Software engineering.
Classification: LCC Q183.9 .S74 2017 | DDC 005.1--dc23
LC record available at https://lccn.loc.gov/2016022277

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

List of Figures

List of Tables

About the Editors

Dr. Jeffrey C. Carver is an associate professor in the Department of Computer Science at the University of Alabama. Prior to his position at the University of Alabama, he was an assistant professor in the Department of Computer Science at Mississippi State University. He earned his PhD in computer science from the University of Maryland. His main research interests include software engineering for science, empirical software engineering, software quality, human factors in software engineering, and software process improvement. He is the primary organizer of the workshop series on Software Engineering for Science (http://www.SE4Science.org/workshops). He is a Senior Member of the IEEE Computer Society and a Senior Member of the ACM. Contact him at carver@cs.ua.edu.

Neil P. Chue Hong is director of the Software Sustainability Institute at the University of Edinburgh, which works to enable the continued improvement and impact of research software. Prior to this he was director of OMII-UK at the University of Southampton, which provided and supported free, open-source software for the UK e-Research community. He has a masters degree in computational physics from the University of Edinburgh and previously worked at Edinburgh Parallel Computing Centre as a principal consultant and project manager on data integration projects. His research interests include barriers and incentives in research software ecosystems and the role of software as a research object. He is the editor-in-chief of the *Journal of Open Research Software* and chair of the Software Carpentry Foundation Advisory Council. Contact him at N.ChueHong@software.ac.uk.

George K. Thiruvathukal is a professor of computer science at Loyola University Chicago and visiting faculty at Argonne National Laboratory in the Math and Computer Science Division and the Argonne Leadership Computing Facility. His research interests include parallel and distributed systems, software engineering, programming languages, operating systems, digital humanities, computational science, computing education, and broadening participation in computer science. His current research is focused on software metrics in open-source mathematical and scientific software. Professor Thiruvathukal is a member of the IEEE, IEEE Computer Society, and ACM.

List of Contributors

Daniel P. Ames
Department of Civil & Environmental Engineering
Brigham Young University
Provo, UT, USA

Katie Antypas
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Satish Balay
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL, USA

Roscoe A. Bartlett
Sandia National Laboratories
Albuquerque, NM, USA

Jed Brown
Department of Computer Science
University of Colorado Boulder
Boulder, CO, USA

Laura Christopherson
RENCI
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA

Ethan Coon
Computational Earth Sciences
Los Alamos National Laboratory
Los Alamos, NM, USA

Alva Couch
Department of Computer Science
Tufts University
Medford, MA, USA

Pabitra Dash
Utah State University
Logan, UT, USA

Anshu Dubey
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL, USA

Daniel Hook
Software Group
ESG Solutions
Kingston, ON, Canada

Jeffery S. Horsburgh
Department of Civil & Environmental Engineering
Utah State University
Logan, UT, USA

Ray Idaszak
RENCI
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA

Arne N. Johanson
Department of Computer Science
Kiel University
Kiel, Germany

Upulee Kanewala
Computer Science Department
Montana State University
Bozeman, MT, USA

Matthew Knepley
Department of Computational & Applied Mathematics
Rice University
Houston, TX, USA

Xiaoye Sherry Li
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Lois Curfman McInnes
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL, USA

Brian Miles
CGI Group Inc.
Fairfax, VA, USA

J. David Moulton
Mathematical Modeling and Analysis
Los Alamos National Laboratory
Los Alamos, NM, USA

Andreas Oschlies
GEOMAR Helmholtz Centre for Ocean Research
Kiel, Germany

Matthew Patrick
Department of Plant Sciences
University of Cambridge
Cambridge, United Kingdom

Katherine Riley
Argonne Leadership Computing Facility

Argonne National Laboratory
Lemont, IL, USA

Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL, USA

Spencer Smith
Computing and Software Department
McMaster University
Hamilton, ON, Canada

Calvin Spealman
Caktus Consulting Group, LLC
Durham, NC, USA

Michael Stealey
RENCI
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA

David G. Tarboton
Department of Civil & Environmental Engineering
Utah State University
Logan, UT, USA

Dali Wang
Climate Change Science Institute
Oak Ridge National Laboratory
Oak Ridge, TN, USA

James M. Willenbring
Sandia National Laboratories
Albuquerque, NM, USA

Frank Winkler
National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Boris Worm
Biology Department
Dalhousie University
Halifax, NS, Canada

Ulrike Meier Yang
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA

Zhuo Yao
Department of Electrical Engineering & Computer Science
University of Tennessee
Knoxville, TN, USA

Hong Yi
RENCI
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA

Acknowledgments

Jeffrey C. Carver was partially supported by grants 1243887 and 1445344 from the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Neil P. Chue Hong was supported by the UK Engineering and Physical Sciences Research Council (EPSRC) Grant EP/H043160/1 and EPSRC, BBSRC and ESRC Grant EP/N006410/1 for the UK Software Sustainability Institute.

George K. Thiruvathukal was partially supported by grant 1445347 from the National Science Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Introduction

General Overview

Scientific software is a special class of software that includes software developed to support various scientific endeavors that would be difficult, or impossible, to perform experimentally or without computational support. Included in this class of software are, at least, the following:

- Software that solves complex computationally- or data-intensive problems, ranging from large, parallel simulations of physical phenomena run on HPC machines, to smaller simulations developed and used by groups of scientists or engineers on a desktop machine or small cluster
- Applications that support scientific research and experiments, including systems that manage large data sets
- Systems that provide infrastructure support, e.g. messaging middleware, scheduling software
- Libraries for mathematical and scientific programming, e.g. linear algebra and symbolic computing

The development of scientific software differs significantly from the development of more traditional business information systems, from which many software engineering best practices and tools have been drawn. These differences appear at various phases of the software lifecycle as outlined below:

- Requirements:
  - Risks due to the exploration of relatively unknown scientific/engineering phenomena
  - Risks due to essential (inherent) domain complexity
  - Constant change as new information is gathered, e.g. results of a simulation inform domain understanding
- Design
  - Data dependencies within the software

  - The need to identify the most appropriate parallelization strategy for scientific software algorithms
  - The presence of complex communication or I/O patterns that could degrade performance
  - The need for fault tolerance and task migration mechanisms to mitigate the need to restart time-consuming, parallel computations due to software or hardware errors

- Coding
  - Highly specialized skill set required in numerical algorithms and systems (to squeeze out performance)
- Validation and Verification
  - Results are often unknown when exploring novel science or engineering areas and algorithms
  - Popular software engineering tools often do not work on the architectures used in computational science and engineering
- Deployment
  - Larger node and core sizes coupled with long runtimes result in increased likelihood of failure of computing elements
  - Long software lifespans necessitate porting across multiple platforms

In addition to the challenges presented by these methodological differences, scientific software development also faces people-related challenges. First, educational institutions teach students high-level languages and programming techniques. As a result, there is a lack of developers with knowledge of relevant languages, like Fortran, or low-level skills to handle tasks like code optimization. Second, the dearth of interdisciplinary computational science programs is reducing the pipeline of graduates who have the experience required to be effective in the scientific software domain. Furthermore, the lack of these programs is reducing the motivation for graduates to pursue careers in scientific software. Third, the knowledge, skills, and incentives present in scientific software development differ from those present in traditional software domains. For example, scientific developers may lack formal software engineering training, trained software engineers may lack the required depth of understanding of the science domain, and the incentives in the science domain focus on timely scientific results rather than more traditional software quality/productivity goals.

The continuing increase in the importance and prevalence of software developed in support of science motivates the need to better understand how software engineering is and should be practiced. Specifically, there is a need to understand which software engineering practices are effective for scientific software and which are not. Some of the ineffective practices may need further refinements to fit within the scientific context. To increase our collective understanding of software engineering for science, this book consists of a collection of peer-reviewed chapters that describe experiences with applying software engineering practices to the development of scientific software.

Publications regarding this topic have seen growth in recent years as evidenced by the ongoing *Software Engineering for Science* workshop series[1] [1–5], workshops on software development as part of the *IEEE International Conference on eScience*[2,3] conference, and case studies submitted to the *Working towards Sustainable Scientific Software: Practice and Experiences* workshop series[4,5]. Books such as *Practical Computing for Biologists* [6] and *Effective Computation in Physics* [8] have introduced the application of software engineering techniques to scientific domains. In 2014, *Nature* launched a new section, *Nature Toolbox*[6], which includes substantial coverage of software engineering issues in research. In addition, this topic has been a longstanding one in *Computing in Science and Engineering* (CiSE)[7], which sits at the intersection of computer science and complex scientific domains, notably physics, chemistry, biology, and engineering. CiSE also has recently introduced a Software Engineering Track to more explicitly focus on these types of issues[8]. EduPar is an education effort aimed at developing the specialized skill set (in concurrent, parallel, and distributed computing) needed for scientific software development [7][9].

In terms of funding, the United States Department of Energy funded the Interoperable Design of Extreme-Scale Application Software (IDEAS) project[10]. The goal of IDEAS is to improve scientific productivity of extreme-scale science through the use of appropriate software engineering practices.

Overview of Book Contents

We prepared this book by selecting the set of chapter proposals submitted in response to an open solicitation that fit with an overall vision for the book.

[1] http://www.SE4Science.org/workshops
[2] http://escience2010.org/pdf/cse%20workshop.pdf
[3] http://software.ac.uk/maintainable-software-practice-workshop
[4] http://openresearchsoftware.metajnl.com/collections/special/working-towards-sustainable-software-for-science/
[5] http://openresearchsoftware.metajnl.com/collections/special/working-towards-sustainable-software-for-science-practice-and-experiences/
[6] http://www.nature.com/news/toolbox
[7] http://computer.org/cise
[8] https://www.computer.org/cms/Computer.org/ComputingNow/docs/2016-software-engineering-track.pdf
[9] http://grid.cs.gsu.edu/ tcpp/curriculum/?q=edupar
[10] http://ideas-productivity.org

The chapters underwent peer review from the editors and authors of other chapters to ensure quality and consistency.

The chapters in this book are designed to be self-contained. That is, readers can begin reading whichever chapter(s) are interesting without reading the prior chapters. In some cases, chapters have pointers to more detailed information located elsewhere in the book. That said, Chapter 1 does provide a detailed overview of the Scientific Software lifecycle. To group relevant material, we organized the book into three sections. Please note that the ideas expressed in the chapters do not necessarily reflect our own ideas. As this book focuses on documenting the current state of software engineering in scientific software development, we provide an unvarnished treatment of lessons learned from a diverse set of projects.

General Software Engineering

This section provides a general overview of the scientific software development process. The authors of chapters in this section highlight key issues commonly arising during scientific software development. The chapters then describe solutions to those problems. This section includes three chapters.

Chapter 1, *Software Process for Multiphysics Multicomponent Codes* provides an overview of the scientific software lifecycle, including a number of common challenges faced by scientific software developers (note readers not interested in the full chapter may find this section interesting). The chapter describes how two projects, the long-running FLASH and newer Amanzi, faced a specific set of these challenges: software architecture and modularization, design of a testing regime, unique documentation needs and challenges, and the tension between intellectual property and open science. The lessons learned from these projects should be of interest to scientific software developers.

Chapter 2, *A Rational Document Driven Design Process for Scientific Software* argues for the feasibility and benefit of using a set of documentation drawn from the waterfall development model to guide the development of scientific software. The chapter first addresses the common arguments that scientific software cannot use such a structured process. Then the chapter explains which artifacts developers can find useful when developing scientific software. Finally, the chapter illustrates the document driven approach with a small example.

Chapter 3, *Making Scientific Software Easier to Understand, Test, and Communicate through Software Engineering* argues that the complexity of scientific software leads to difficulties in understanding, testing, and communication. To illustrate this point, the chapter describes three case studies from the domain of computational plant biology. The complexity of the underlying scientific processes and the uncertainty of the expected outputs makes adequately testing, understanding, and communicating the software a challenge. Scientists who lack formal software engineering training may find these challenges especially difficult. To alleviate these challenges, this chapter reinterprets two testing techniques to make them more intuitive for scientists.

Software Testing

This section provides examples of the use of testing in scientific software development. The authors of chapters in this section highlight key issues associated with testing and how those issues present particular challenges for scientific software development (e.g. test oracles). The chapters then describe solutions and case studies aimed at applying testing to scientific software development efforts. This section includes four chapters.

Chapter 4, *Testing of Scientific Software: Impacts on Research Credibility, Development Productivity, Maturation, and Sustainability* provides an overview of key testing terminology and explains an important guiding principle of software quality: understanding stakeholders/customers. The chapter argues for the importance of automated testing and describes the specific challenges presented by scientific software. Those challenges include testing floating point data, scalability, and the domain model. The chapter finishes with a discussion of test suite maintenance.

Chapter 5, *Preserving Reproducibility through Regression Testing* describes how the practice of regression testing can help developers ensure that results are repeatable as software changes over time. Regression testing is the practice of repeating previously successful tests to detect problems due to changes to the software. This chapter describes two key challenges faced when testing scientific software, the oracle problem (the lack of information about the expected output) and the tolerance problem (the acceptable level of uncertainty in the answer). The chapter then presents a case study to illustrate how regression testing can help developers address these challenges and develop software with reproducible results. The case study shows that without regression tests, faults would have been more costly.

Chapter 6, *Building a Function Testing Platform for Complex Scientific Code* describes an approach to better understand and modularize complex codes as well as generate functional testing for key software modules. The chapter defines a *Function Unit* as a specific scientific function, which may be implemented in one or more modules. The *Function Unit Testing* approach targets code for which unit tests are sparse and aims to facilitate and expedite validation and verification via computational experiments. To illustrate the usefulness of this approach, the chapter describes its application to the Terrestrial Land Model within the Accelerated Climate Modeling for Energy (ACME) project.

Chapter 7, *Automated Metamorphic Testing of Scientific Software* addresses one of the most challenging aspects of testing scientific software, i.e. the lack of test oracles. This chapter first provides an overview of the test oracle problem (which may be of interest even to readers who are not interested in the main focus of this chapter). The lack of test oracles, often resulting from the exploration of new science or the complexities of the expected results, leads to incomplete testing that may not reveal subtle errors. Metamorphic testing addresses this problem by developing test cases through metamorphic relations. A metamorphic relation specifies how a particular change to the input should change the output. The chapter describes a machine learning approach to automatically predict metamorphic relations which can then serve as test oracles. The chapter then illustrates the approach on several open source scientific programs as well as on in-house developed scientific code called SAXS.

Experiences

This section provides examples of applying software engineering techniques to scientific software. Scientific software encompasses not only computational modeling, but also software for data management and analysis, and libraries that support higher-level applications. In these chapters, the authors describe their experiences and lessons learned from developing complex scientific software in different domains. The challenges are both cultural and technical. The ability to communicate and diffuse knowledge is of primary importance. This section includes three chapters.

Chapter 8, *Evaluating Hierarchical Domain-Specific Languages for Computational Science: Applying the Sprat Approach to a Marine Ecosystem Model* examines the role of domain-specific languages for bridging the knowledge transfer gap between the computational sciences and software engineering. The chapter defines the Sprat approach, a hierarchical model in the field of marine ecosystem modeling. Then, the chapter illustrates how developers can implement scientific software utilizing a multi-layered model that enables a clear separation of concerns allowing scientists to contribute to the development of complex simulation software.

Chapter 9, *Providing Mixed-Language and Legacy Support in a Library: Experiences of Developing PETSc* summarizes the techniques developers employed to build the PETSc numerical library (written in C) to portably and efficiently support its use from modern and legacy versions of Fortran. The chapter provides concrete examples of solutions to challenges facing scientific software library maintainers who must support software written in legacy versions of programming languages.

Chapter 10, *HydroShare — A Case Study of the Application of Modern Software Engineering to a Large, Distributed, Federally-Funded, Scientific Software Development Project* presents a case study on the challenges of introducing software engineering best practices such as code versioning, continuous integration, and team communication into a typical scientific software development project. The chapter describes the challenges faced because of differing skill levels, cultural norms, and incentives along with the solutions developed by the project to diffuse knowledge and practice.

Key Chapter Takeaways

The following list provides the key takeaways from each chapter. This list should help readers better understand which chapters will be most relevant to their situation. As stated earlier, the takeaways from each chapter are the opinions of the chapter authors and not necessarily of the editors.

Chapter 1

- The development lifecycle for scientific software must reflect stages that are not present in most other types of software, including model development, discretization, and numerical algorithm development.
- The requirements evolve during the development cycle because the requirements may themselves be the subject of the research.
- Modularizing multi-component software to achieve separation of concerns is an important task, but it difficult to achieve due to the monolithic nature of the software and the need for performance.
- The development of scientific software (especially multiphysics, multidomain software) is challenging because of the complexity of the underlying scientific domain, the interdisciplinary nature of the work, and other institutional and cultural challenges.
- Balancing continuous development with ongoing production requires open development with good contribution and distribution policies.

Chapter 2

- Use of a rational document-driven design process is feasible in scientific software, even if rational documentation has to be created post hoc to describe a development process that was not rational.
- Although the process can be time consuming, documenting requirements, design, testing and artifact traceability improves software quality (e.g., verifiability, usability, maintainability, reusability, understandability, and reproducibility).
- Developers can integrate existing software development tools for tasks like version control, issue tracking, unit testing, and documentation generation to reduce the burden of performing those tasks.

Chapter 3

- Scientific software is often difficult to test because it is used to answer new questions in experimental research.

- Scientists are often unfamiliar with advanced software engineering techniques and do not have enough time to learn them, therefore we should describe software engineering techniques with concepts more familiar to scientists.

- Iterative hypothesis testing and search-based pseudo-oracles can be used to help scientists produce rigorous test suites in the face of a dearth of a priori information about its behavior.

Chapter 4

- The complexity of multiphysics scientific models and the presence of heterogeneous high-performance computers with complex memory hierarchies requires the development of complex software, which is increasingly difficult to test and maintain.

- Performing extensive software testing not only leads to software that delivers more correct results but also facilitates further development, refactoring, and portability.

- Developers can obtain quality tests by using granular tests at different levels of the software, e.g., fine-grained tests are foundational because they can be executed quickly and localize problems while higher-level tests ensure proper interaction of larger pieces of software.

- Use of an automated testing framework is critical for performing regular, possibly daily, testing to quickly uncover faults.

- Clearly defined testing roles and procedures are essential to sustain the viability of the software.

Chapter 5

- Use of regular, automated testing against historical results, e.g., regression testing, helps developers ensure reproducibility and helps prevent the introduction of faults during maintenance.

- Use of regression testing can help developers mitigate against the oracle problem (lack of information about the expected output) and the tolerance problem (level of uncertainty in the output).

Chapter 6

- The use of a scientific function testing platform with a compiler-based code analyzer and an automatic prototype platform can help developers test large-scale scientific software when unit tests are sparse.

- The function testing platform can help model developers and users better understand complex scientific code, modularize complex code, and generate comprehensive functional testing for complex code.

Chapter 7

- The oracle problem poses a major challenge for conducting systematic automated testing of scientific software.
- Metamorphic testing can be used for automated testing of scientific software by checking whether the software behaves according to a set of metamorphic relations, which are relationships between multiple input and output pairs.
- When used in automated unit testing, a metamorphic testing approach is highly effective in detecting faults.

Chapter 8

- Scientists can use domain-specific languages (DSLs) to implement well-engineered software without extensive software engineering training.
- Integration of multiple DSLs from different domains can help scientists from different disciplines collaborate to implement complex and coupled simulation software.
- DSLs for scientists must have the following characteristics: appropriate level of abstraction for the meta-model, syntax that allows scientists to quickly experiment, have tool support, and provide working code examples as documentation.

Chapter 9

- Multi-language software, specifically Fortran, C, and C++, is still important and requires care on the part of library developers, benefitting from concrete guidance on how to call Fortran from C/C++ and how to call C/C++ from Fortran.
- Mapping of all common C-based constructs in multiple versions of Fortran allows developers to use different versions of Fortran in multi-language software.

Chapter 10

- Use of modern software engineering practices helps increase the sustainability, quality and usefulness of large scientific projects, thereby enhancing the career of the responsible scientists.
- Use of modern software engineering practices enables software developers and research scientists to work together to make new and valuable contributions to the code base, especially from a broader community perspective.
- Use of modern software engineering practices on large projects increases the overall code capability and quality of science results by propagating these practices to a broader community, including students and post-doctoral researchers.