SOFTWARE ENGINEERING FOR SCIENCE

This chapter distributed electronically in accordance with Author Addendum to Publication Agreement

SPARC Author Addendum 3.0. www.arl.org/sparc/

Cite as:
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 CONCURRENY 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
MATLAB® is a trademark of The MathWorks, Inc. and is used with permission. The MathWorks does not warrant the accuracy of the text or exercises in this book. This book’s use or discussion of MATLAB® software or related products does not constitute endorsement or sponsorship by The MathWorks of a particular pedagogical approach or particular use of the MATLAB® software.

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

© 2017 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper
Version Date: 20160817

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

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data
Title: Software engineering for science / edited by Jeffrey Carver, Neil P. Chue Hong, and George K. Thiruvathukal.
Identifiers: LCCN 2016022277 | ISBN 9781498743853 (alk. paper)
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
Chapter 10

HydroShare – A Case Study of the Application of Modern Software Engineering to a Large Distributed Federally-Funded Scientific Software Development Project

Ray Idaszak, David G. Tarboton (Principal Investigator), Hong Yi, Laura Christopherson, Michael J. Stealey, Brian Miles, Pabitra Dash, Alva Couch, Calvin Spealman, Jeffery S. Horsburgh, and Daniel P. Ames

10.1 Introduction to HydroShare ............................................. 218
10.2 Informing the Need for Software Engineering Best Practices for Science ......................................................... 220
10.3 Challenges Faced and Lessons Learned .............................. 221
  10.3.1 Cultural and Technical Challenges .............................. 221
  10.3.2 Waiting Too Long between Code Merges ...................... 223
  10.3.3 Establishing a Development Environment .................... 224
10.4 Adopted Approach to Software Development Based on the Lessons Learned .................................................. 224
  10.4.1 Adopting Best Practices in Modern Software Engineering ................................................................. 225
  10.4.2 Iterative Software Development .................................. 226
  10.4.3 Virtual Machines ..................................................... 227
  10.4.4 Code Versioning ...................................................... 228
  10.4.5 Code Reviews ......................................................... 228
  10.4.6 Testing and Test-Driven Development ......................... 229
  10.4.7 Team Communication ............................................... 229
  10.4.8 DevOps ............................................................... 230
10.5 Making Software Engineering More Feasible and Easier to Integrate into One’s Research Activities ..................... 231
10.6 Conclusion ..................................................................... 232
Acknowledgments .............................................................. 233
Abstract

HydroShare is an online collaborative system under development to support the open sharing of hydrologic data, analytical tools, and computer models. With HydroShare, scientists can easily discover, access, and analyze hydrologic data and thereby enhance the production and reproducibility of hydrologic scientific results. HydroShare also takes advantage of emerging social media functionality to enable users to enhance information about and collaboration around hydrologic data and models.

HydroShare is being developed by an interdisciplinary collaborative team of domain scientists, university software developers, and professional software engineers from ten institutions located across the United States. While the combination of non–co-located, diverse stakeholders presents communication and management challenges, the interdisciplinary nature of the team is integral to the project’s goal of improving scientific software development and capabilities in academia.

This chapter describes the challenges faced and lessons learned with the development of HydroShare, as well as the approach to software development that the HydroShare team adopted on the basis of the lessons learned. The chapter closes with recommendations for the application of modern software engineering techniques to large, collaborative, scientific software development projects, similar to the National Science Foundation (NSF)–funded HydroShare, in order to promote the successful application of the approach described herein by other teams for other projects.

10.1 Introduction to HydroShare

The HydroShare software development project is funded by the National Science Foundation (NSF) through its Software Infrastructure for Sustained Innovation program [333, 336]. Domain scientists, professional\(^1\) software engineers, and academic software developers from ten academic institutions located across the United States\(^2\) collaborate to develop HydroShare—an online,

\(^1\)The term professional, as used here refers to an individual that has received formal education on software development and has applied this knowledge in a commercial or equivalent context.

\(^2\)Brigham Young University, Caktus Group, Consortium of Universities for the Advancement of Hydrologic Science, Inc., Purdue University, Renaissance Computing Institute (RENCI) at the University of North Carolina at Chapel Hill, Tufts University, Institute for the Environment at the University of North Carolina at Chapel Hill, University of Texas at Austin, University of Virginia, and Utah State University.
collaborative system that extends the data-sharing capabilities of the Hydrologic Information System (HIS), which was developed by the Consortium of Universities for the Advancement of Hydrologic Sciences, Inc. (CUAHSI) [355]. HydroShare extends the data-sharing capabilities of HIS by broadening the classes of data that are accommodated, enabling the sharing of computer models and model components, and incorporating social media functionality in order to enhance communication and collaboration around hydrologic data and models [350, 351, 353].

In cooperation with CUAHSI, HydroShare is being used by the National Flood Interoperability Experiment (NFIE), which is a collaboration between the National Weather Service, government and commercial partners, and the academic community. NFIE is working to build a next-generation, high-resolution, near–real-time hydrologic simulation and forecasting model for the United States. With HydroShare, NFIE is able to better facilitate the flow of information between the federal, state, and local entities responsible for flood measurement, forecasting, and planning [338]. This near–real-time information also can be used by first responders during severe weather events to navigate to people in need of assistance [339].

The HydroShare project provides an example of the application of modern software engineering techniques to the development of scientific software. At the project’s outset, most members of the HydroShare team did not fully understand the difference between software development and software engineering, nor were they familiar with iterative software methodology, code refactoring, continuous integration, or test-driven development (explained in Section 10.4.6). Much of the functionality of HydroShare—such as user interface, access control, social media incorporation, metadata handling, search and discovery, analytics, simulation, and storage capabilities—also was challenging for the team. While many members of the team had previous experience in the software development of hydrologic models, including models containing very complex algorithms and data structures, none of the models that had been developed by team members had the depth or complexity of the HydroShare software stack, and none required distributed code development and coordination across a large team. Thus, the team quickly realized the need to apply modern software engineering practices as part of the HydroShare experience. At the time of this writing, four years into the project, the team is now capable of applying advanced software engineering techniques to the development of HydroShare.

This chapter describes the approach, experience, and lessons learned when applying modern software engineering techniques to a large scientific software project, HydroShare. Recommendations are provided for how to integrate best practices in modern software engineering into large, collaborative research projects such as HydroShare. The overall intent is to support the advancement of science and expand the use of sustainable software engineering practices in academia. The goal is for other scientific software development teams to be able to adopt and adapt the techniques and practices described in this chapter.
10.2 Informing the Need for Software Engineering Best Practices for Science

Modern scientific research relies on software. Software enables scientists to collect data, perform analyses, run numerical and statistical models, and visualize data. With the aid of software, scientists are able to answer key research questions and test hypotheses that can revolutionize what is known about the world. Life-sustaining policies, products, and techniques—such as clinical therapies, pharmaceutical compounds, and solutions to environmental problems—derive from software-enabled scientific research.

Software such as HydroShare that supports data collection, analysis, and modeling is often used to accomplish research goals. Hannay, MacLeod, and Singer [349] have noted that scientists spend as much as 40% of their time using software. Often, existing software is ill-suited to a particular research project or, in the case of commercial software, prohibitively expensive. The result is that scientists often develop their own software—spending as much as 30% of their time doing so [349]—even though few incentives exist for software development in traditional tenure and promotion decision-making processes [352]. In other words, the time that an academic scientist spends developing software is not rewarded or recognized as a significant, independent accomplishment. Tenure and promotion, for example, are based on influential research, a successful publication record, the acquisition of grants, and teaching—not on whether one can author good software. Additionally, many funding agencies wish to see their funds going toward time spent on traditional research activities, not writing software.

While not incentivized, academic scientists continue to develop their own software. However, most academic scientists are not trained in software development or software engineering [342, 345, 359]. Software development courses, typically offered by computer science departments, are not required for most non-majors. Additionally, the training that scientists do receive from computer science departments often is perceived as overly general or abstract, and scientists may not see the relevance of such training [349]. As a result of the lack of training in software development and software engineering, the software that is developed by academic scientists often is not built to the development standards of the commercial sector. Software engineering best practices, such as documentation, versioning, and testing, may not be applied during the creation of academic scientific software. Furthermore, most academic software is developed to suit the needs of a specific research project and thus may not be applicable to other research projects or sustainable beyond the life of the initial project.

The lack of training in software development and software engineering can have dire consequences [346]. For instance, software that is developed without the use of proven software engineering techniques may lead to errors in
the code. Even minor errors influence the validity of research findings; indeed, in some cases, papers have been retracted from scientific journals and careers have been ruined [357]. Paper rejections and irreproducible results due to poor-quality software impede the advancement of science and impart huge financial repercussions. Under the worst case scenario, programming errors can lead to the loss of lives if erroneous findings result in faulty medical technologies or misdirected policies on disaster response, to provide examples.

The detection of errors in academic software is extremely challenging, however. While manuscripts submitted for journal publication must undergo a peer review process, the software code that is used to generate the findings presented in manuscripts is rarely subjected to a peer review process or other measures of quality assurance. Yet, peer review and testing of software code are critical for the credibility of science and require software engineering best practices.

Of significance, the risk of introducing error into scientific research through the use of low-quality software provides a little recognized, but highly impactful, incentive for the adoption of software engineering best practices in academic scientific software development.

The HydroShare project addresses the challenges and highlights the benefits of the adoption of software engineering best practices through a collaborative scientific software project involving a large, geographically dispersed team of academic scientists, academic software developers, and professional software engineers.

10.3 Challenges Faced and Lessons Learned

This section describes the challenges faced and lessons learned when applying modern software engineering best practices to a software development project in hydrology. Modern software engineering, as used here refers to “the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software” [330].

10.3.1 Cultural and Technical Challenges

Early on, the HydroShare team identified several overarching culture challenges. First, the team found that it is not a tradition to use modern software engineering best practices in the development of academic software due to a lack of incentives and a failure to recognize the benefits, as discussed above. The perception was that good software engineering practices are not needed to obtain scientific results and to publish scientific papers. Second, graduate students often develop software for their faculty advisors, yet graduate stu-
students have very short-term goals (i.e., graduate in the next couple of years), so software sustainability is not a high priority. Third, graduate students and their faculty advisors typically have not received formal training in software development, let alone software engineering. Fourth and lastly, the rigorous metadata requirements necessary for reproducible science make scientific software systems more complex than other types of software and thus require significant time to create unit tests. This presents a paradox, as the more complex software is, the more benefit one gets from having comprehensive unit tests.

The team also encountered more specific technical challenges. For example, as implementation of our HydroShare project began, the team quickly realized that most members were not familiar with Git, GitHub, or continuous integration (i.e., a development practice that requires developers to integrate code into a shared repository on a very frequent basis). The decision was thus made to assign only members at the lead technical institution the task of implementing initial beta release functionalities in order to expedite creation of the code infrastructure for subsequent collaborative development and continuous integration by the broader team members. However, this limited HydroShare beta release functionality to only those functionalities that could be implemented by the lead technical institution. This approach did expedite the initial release of the system, but the approach also precluded the ability for other team members to contribute to the development. For HydroShare, this trade-off was acceptable as the other team members used the additional time to get versed on continuous integration, Git, GitHub, and other specific technologies and approaches used in HydroShare software development.

Early on in the project, the team held several in-person meetings, as well as weekly team teleconferences, that served to achieve the development objectives, including the development of a data model (i.e., a conceptual model of how data elements relate to each other) and access control policies (i.e., policies to restrict access to data) and thorough consideration of how to accommodate hydrologic models within HydroShare. As implementation progressed and software engineering principles, such as code versioning (i.e., management of revisions to source code) and continuous integration, were diffused from the professional software engineers to the hydrologists, additional challenges emerged. For example, the distributed development team experienced difficulty achieving short-release cycles of continuous integration of the Django-based system using Git and GitHub. Django is a large, complex, open source, python-based web development framework, in which its customization model and content data are stored in databases [328]. Django proved to be difficult to manage via version control by a team with members of various skill levels. Specifically, the challenge was how to manage multiple, distributed development teams that were simultaneously checking out their own branch\(^3\) of HydroShare, while

\(^3\)A branch in GitHub lives separately from the production codebase, thus allowing for experimentation without affecting the master branch (or production codebase).
HydroShare – A Case Study of the Application

maintaining consistency in the back-end Django database. This led to multiple, complex code feature branches being checked out and worked on at the same time—without a sufficient number of intermediate merges.

Described below are two specific challenges—waiting too long between code merges and establishing a development environment—examined in greater depth, including the downstream challenges and lessons learned.

10.3.2 Waiting Too Long between Code Merges

To highlight the complications that may arise from waiting too long between code merges, this section considers a release of HydroShare in which two key items were addressed on different branches: base class refactoring and a change in the approach to access control. This presented a non-trivial challenge because of the intertwining of these two items, along with the need to preserve existing resources.

The HydroShare base class refactoring branch endeavored to promote the HydroShare Generic Resource type functionality from being abstract to fully defined. Being able to extend upon a fully defined resource opened the door for easier searching, indexing, and database querying that wouldn’t otherwise be possible if the team had kept extending from the previously defined abstract model. Once this was implemented and tested for the Generic Resource type, the team then needed to apply this to all of the other HydroShare resource types in close coordination with the developers that had originally created them in order to ensure against loss of context, extended metadata, or other resource-specific attributes.

The HydroShare access control branch endeavored to implement an access control model that the team designed to best suit the hydrology research community [347]. However, this uniquely designed HydroShare access control model meant that it was necessarily non-standard and non-compliant with basic use of Django; thus, the team took extra care in how this change was implemented. The first, albeit incomplete, implementation of the HydroShare access control was integrated on top of Django’s permission system for the sake of simplicity and the need to get an initial working version of HydroShare. To implement the full HydroShare access control, the team needed to decouple from Django’s permission system and enforce a HydroShare-specific model, thereby adding additional system complexity.

To further complicate things, the HydroShare access control and base class refactoring had to be implemented on top of existing HydroShare resources in production use. The integrated rule-oriented data system (iRODS) [334] is used as a data management back-end to HydroShare. The challenge however, was migrating all of the existing HydroShare resources that were in use by users when the new resource and access control schema didn’t fit the existing iRODS storage policies. Multiple steps and operations on the resources and database were required to properly migrate resources into the new models and access control schema. This proved to be quite a challenging endeavor.
Each of these items on their own presented a significant task; however, the summation of all of these branches into a single release required numerous dry-runs and small intermediate tests based on the results of the dry-runs before the team was confident that it was right. The team put as much time into testing and validation as they did into coding the changes themselves. The main lesson learned from this experience is that it is best to perform smaller, but more frequent merges, rather than a large release with multiple complex merges. With the former approach, the merge complexity will be reduced and time will be saved.

10.3.3 Establishing a Development Environment

Another major challenge for the development team was setting up the integrated development environment for individual developers. This presented a challenge mainly due to the many Docker containers that the HydroShare system uses, as well as the fact that most of the developers did not have knowledge of Docker configuration, which was a relatively new technology at the beginning of the HydroShare project. This challenge was resolved by scripting the development environment, such that with few commands, the team could quickly set up the HydroShare development environment—something that had previously taken hours. As the development team was distributed, weekly videoconferences were used to train new HydroShare developers on how to set up the development environment.

The HydroShare software developers faced major challenges in code contribution in the early stages of the HydroShare project due to the size of the project and their inexperience, particularly when working in a distributed team environment. In addition, the team didn’t have procedures in place for how to effectively contribute code using GitHub (also discussed in Section 10.4.8), which was new to many team members. In order to solve these challenges, the team created very detailed documentation specific to the project on how to push/pull to/from GitHub. In addition, hands-on training was provided to all software developers on best practices for using GitHub. In order to improve code quality, the team adopted the GitHub pull request feature for code review, whereby all code had to be code reviewed by an independent team member prior to merging the pull request. We found these practices to be extremely beneficial in providing the momentum to establish our software development environment.

10.4 Adopted Approach to Software Development Based on the Lessons Learned

This section conveys the approach to software development that was adopted for HydroShare based on the lessons learned early on in the project.
The approach includes the adoption of an iterative approach to software development that incorporates best practices in modern software engineering. Highlighted are several best practices in software engineering, including the use of virtual machines, code versioning, code reviews, and test-driven development. This section concludes with a discussion of the role and importance of communication and DevOps in facilitating effective multi-institutional collaboration.

10.4.1 Adopting Best Practices in Modern Software Engineering

One of the goals of the HydroShare Project is to continually adopt modern software engineering techniques to all scientific software development efforts. Although a scientist can author high-value software code, s/he approaches software development as a means to an end, with the end being new research findings. A software engineer, in contrast, approaches software development with code quality and sustainability as primary goals—not application. To a scientist, the research process is emphasized, and the final product is a set of scientific findings, which should be accurate, reproducible, and generalizable. To a software engineer, the coding process is emphasized, and the software code is the product, which should be error-free and reusable for solving other problems. In the same way that a scientist carefully designs a study to answer a research question or test a hypothesis, a software engineer carefully designs the code s/he will write to create new functionality. For example, software engineers use design patterns, or reusable and generalizable units of code that solve common software problems. Most scientists are not familiar with the concept of design patterns. Instead of combining reusable, tested units of code into new software, scientists often choose to write code from scratch in order to address a specific research need; after that need has been met, the software code is often tossed aside. Scientists are not as concerned about ensuring that the code is free of bugs because the code is not the object of interest, so code testing is not common practice. Software engineers, however, are trained to ensure that the code is free of bugs because quality code is the object of interest, so testing the code for accuracy is common practice.

One could argue that if scientists had lavish funding, they could hire professional software engineers to develop higher quality code over a more expeditions timeline. However, while an abundance of funding is always desirable, this would prevent the realization of certain opportunities. For HydroShare the involvement of hydrologists, including both graduate students and their faculty advisors, in software coding was extremely important for several reasons:

- As subject matter experts, the scientists were able to illuminate salient uses cases.
As co-creators and owners of the software, the scientific community will be more likely to adopt the software and shepherd it throughout its lifetime.

As graduate students, the incorporation of modern software engineering practices into their training is imperative in order to better prepare the next generation of hydrologists.

The key is that HydroShare should not be viewed simply as a software artifact, but also as a project that captures human and scientific capital for the advancement of transformative science through mature software engineering methodology. This is an important point. A lot of new ideas and thought processes have been created in the minds of the HydroShare team (i.e. human capital) as a result of this project, and these need to be kept concomitant with the software. Modern software engineering, in part, helps us achieve this.

10.4.2 Iterative Software Development

The Waterfall approach to software development emphasizes a discrete planning phase that includes gathering all possible requirements before the coding phase commences [340]. After a Waterfall phase is concluded, the rule-of-thumb is that that phase should not be revisited. This type of approach does not recognize or make allowances for the unknown and unpredictable. In other words, the Waterfall approach does not provide flexibility regarding changes in requirements, new needs or system uses, or changes in project focus.

Brooks [343] claims that software engineers should be open and ready to throw out unworkable ideas. “The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers” [344]. When a distributed software development team endeavors to integrate several new and existing software systems at the onset of a project, complications can arise that preclude the ability of the team to efficiently and/or practically overcome those challenges. This is especially true in academic software development projects that have limited time and funding and include team members with varying levels of skill. With HydroShare, the team was not exempt from the “throwaway principle” and indeed had to completely discard a well-developed early version of the software due to unforeseen complications with the integration of disparate software systems. This was the result of several factors:

1. The decision to go with a seemingly appropriate technology, with much community adoption in other circles, was flawed at the beginning and made by a subset of the team without full consideration by the broader team. A more inclusive decision process would have led to better articulation regarding the platform requirements and a better outcome.

2. The system that was chosen, while having widespread community adoption in other circles, was one in which the team had no expertise. The
learning curve proved to be too high for them, at least on a practical level and given the time constraints.

3. The team’s lack of expertise was magnified when the team members that made the decision to adopt the system left the project and a new lead development team came onboard without any prior knowledge of the selected technology or understanding of the previous team’s activities; this challenge was exacerbated by lack of transition documentation to guide the new team.

The team has since adopted a more flexible iterative approach with HydroShare, one that embraces change. The conclusion is that one should expect to throw out an early version of a software product and learn from the experience. Also, one should realize that it is so much more efficient (and easier to accept) if this is part of the team’s plan from the start, for when planning to throw out an early version of developed software, a team can view the experience as an exceptional opportunity to learn what works and what doesn’t from the perspectives of software and technology integration, team communication, meeting productivity, and process efficiency. The HydroShare team also found it beneficial to encapsulate functionality in small, loosely coupled systems. For example, the distributed data management system used by HydroShare can work separately from the content management system, which can work separately from the web applications system, and so forth. In the first iteration, the team found that the integration of systems too tightly presents limitations. Unforeseen challenges arise in every software development project; the key is to plan for this early on and in every facet of the project—and expect to throw away at least one early product.

10.4.3 Virtual Machines

The HydroShare team uses virtual machines (VM) in testing and production in order to facilitate the distributed team’s concurrent prototyping and development of the many diverse features of HydroShare. VMs can be created and spun-up very quickly, with configurable memory, processor, disk storage, and operating system to meet the diverse and evolving project and feature requirements. For features that are complex and highly susceptible to error, the HydroShare team creates a VM to test the feature. The team also creates feature-testing VMs for contextually-related features. For example, the group working on the search and filtering functionality has their own VM; the federated identity management group has its own VM; the user interface group has their own VM, and so on. Git (i.e., a revision control system) and GitHub (i.e., a hosting service for Git repositories) are used to manage and eventually merge the work on these VMs into a production release. Generally, a given branch of code that requires testing and feedback from the team is given its own VM. The exception is that some Git branches—especially those for general fixes—don’t require deployment to a VM since they don’t intertwine with
other parts of the system and can be tested locally. Production VMs share an allocation of fifty terabytes of project disk space and another fifty terabytes of replicated disk space located four miles away in order to ensure fault tolerance and disaster recovery.

10.4.4 Code Versioning

Code versioning is a must for any modern software development project, academic or otherwise. There are several popular code versioning systems. The HydroShare team chose Git due to its ability to support distributed development workflows. Unlike other version control systems, Git allows developers to clone the main code repository on their local machines and develop and experiment with new code safely, in an independent environment completely separated from the main codebase. New code can then be submitted for inclusion into the main codebase after being reviewed and tested by other members of the team. This enforces code review, allows for experimentation within a safety net, and enables concurrent streams of development for a speedier process.

10.4.5 Code Reviews

With HydroShare, code reviews have opened up the reading of code and stimulated discussion around the structure of the code—something that was not happening before the team implemented the code review process. However, the team took a while to acclimate to the code review process, whereby the person who reviews the code is always different from the person who authors the code. For HydroShare, a code review includes an evaluation of:

- How well the new units of code address the associated use case;
- Code quality, in terms of clarity, concision, lack of redundancy, and thorough inline documentation;
- How easy the code’s functionality is to use; and
- How the code fared in unit testing (i.e., tests of individual modules written for a particular function that together comprise a larger set of code).

The HydroShare team has found code reviews to be beneficial for encouraging discussion between scientists and software engineers around the structure of the code. These discussions have served as a vehicle for teaching software engineering best practices to the scientists involved in the project, particularly graduate students who are not required to take programming classes as part of their graduate work. In addition to these benefits, estimates suggest that rigorous code review can remove up to 90% of errors from a software product before any code testing is initiated [348] (code testing is discussed in next section).

The key point is that while it is necessary to establish regular code reviews early on, a period of acclimation should be expected.
10.4.6 Testing and Test-Driven Development

The testing of all code prior to release is extremely important for writing sustainable code – code that lasts over time because it has been tested for defects and performs consistently through new releases. There are a variety of tests that may be conducted during development, for example, unit testing, feature testing, regression testing, etc.

Unit testing is used to verify that the code does what it is expected to do without error. Ideally, using the software engineering concept of Test-Driven Development (TDD) [337], the test is written before the code is written. This forces the developer to think more carefully about the structure of the code, consider the best ways to satisfy the expectations of the unit of code, and plan for any error conditions before the code is written.

The HydroShare team has tied unit testing to Jenkins, which is an open source, continuous integration tool [335]. Jenkins is used to implement continuous integration by automating runs of unit tests for both new code submissions and nightly builds of the main codebase. Unit testing is beneficial because it allows developers to test new features within the context of the existing code prior to inclusion in the main codebase. This is done to verify that a new feature will not cause existing tests to fail after it is integrated into the main codebase. When many features are merged, they are tested together in order to ensure that their interactions do not cause failures. Roughly every two to three weeks, the development branch is merged into the production codebase (or master branch), which is the code that runs on the publicly visible HydroShare production site [332]. In this way, new functionality is both adequately reviewed and tested, as well as rapidly released.

While TDD provides an important model for software development, the HydroShare team implemented a hybrid approach by authoring some unit tests after functional HydroShare code was written. This approach was prompted by time constraints and the fact that TDD has a steep learning curve that may cause an initial decrease in developer productivity [356]. Indeed, even at this writing, the HydroShare team is still acclimating to the TDD process. Moreover, the HydroShare team does not yet use TDD for development of user interfaces, as the integration of emulated user interface actions, combined with all relevant user traversals of the HydroShare web page environment, is currently a prohibitively complex development endeavor for a project of the scale and complexity of HydroShare. Testing, combined with thorough initial design, has been shown to result in approximately 40% fewer defects compared to code developed with more ad-hoc testing [358]. The HydroShare team continues to strive toward more comprehensive use of TDD.

10.4.7 Team Communication

Invariably, a new project will commence with a series of meetings. Among the topics of those meetings should be the plan for both team communication
and the software development infrastructure (i.e., the software and hardware used for development). With HydroShare, the establishment of communication protocols and development infrastructure early on in the project supported collaboration and productivity and likely will continue to serve the team well throughout the lifetime of the project.

For example, for weekly meetings of distributed team members, the team employs videoconferencing software with screen sharing capability. For communication outside of meetings, a team email list is used. HipChat [331], a synchronous chat tool, was adopted as a place solely for development-centric discussion, so as to avoid overloading subject matter experts (i.e., domain scientists who do not participate in development) with extraneous information or noise that only serves to distract from the research process. Furthermore, the team adopted a content management system to host all documents for the project, including meeting notes, presentations, use cases, architectural diagrams, API documentation, policies, etc. The team also uses email lists to disseminate community announcements (e.g., announce@hydroshare.org, support@hydroshare.org) and to allow people to obtain support for HydroShare. To describe the project to interested parties, the team has created public-facing web pages. Each of these activities has proven important to the success of HydroShare.

10.4.8 DevOps

In addition to effective communication among team members, close collaboration is essential. Development Operations or DevOps is an industry concept that can be defined as an approach to software development that emphasizes the importance of collaboration between all stakeholders [327]. DevOps recognizes that stakeholders (e.g., programmers, scientists) do not work in isolation. This principle was adopted for HydroShare; software developers and domain scientists work together, closely and continuously, in the development of the HydroShare code. For HydroShare, a software engineer was selected to fill the DevOps lead role because s/he must be a maestro of Git, GitHub, and coding, and few team scientist-developers were skilled with modern software engineering techniques at the start of the project. The appointment of an experienced software engineer as the DevOps lead allows the scientist-developers to learn tools such as Git as they develop and contribute code. The DevOps lead facilitates this learning process by writing task automation scripts in order to simplify and optimize code contributions in Git. With HydroShare, GitHub is used for issue tracking in order to drive new development or track defects (i.e., bugs). GitHub issues are also used to track the progress of code reviews, with developers giving a simple “+1” to indicate that the code has been reviewed and that the DevOps lead may proceed with a code merge. Task automation scripts help the DevOps lead groom the code repository and make Git’s branching and merging processes more transparent. Together, these activities contribute to the DevOps lead’s ability to successfully ensure continuous in-
10.5 Making Software Engineering More Feasible and Easier to Integrate into One’s Research Activities

Many research projects do not have sufficient funding to support training in software development and the fundamentals of good software engineering [345]. Moreover, rigid or process-heavy software development approaches have been shown to be unappealing to scientists [345]. Thus, accepted software engineering approaches to the design, development, documentation, testing, and review of code, for example, may not be employed by scientists. The result is software that is not sustainable or usable by others.

In order to infuse the scientific community with good software engineering practices, it is important to make software engineering practices more appealing to scientists. One approach to encourage the adoption of modern software engineering practices is to emphasize the end result: software that is useful, high quality, and sustainable [341].

Through the HydroShare project, an approach has been identified to integrate software engineering best practices into a large, distributed scientific software development project in a manner that is feasible for scientists. Provided below are several specific recommendations for integrating software engineering practices into one’s research activities.

First, an initial design specification should be completed at the very beginning of a project, followed by an iterative design review for continuous refinement throughout the project development cycle. This software engineering practice increases both software quality and productivity. The initial design should be just enough to get project development going. The design should be reviewed iteratively for continuous refinement as project development advances. The initial minimal set of specifications provides sufficient constraint and guidance for the first iteration of software development in order to ensure that no time is wasted in the present producing a specification that would be changed or abandoned later (especially if one plans to “throw one away” as covered in Section 10.4.2 herein). The design specification then evolves in conjunction with software development to best serve its purpose of guiding and planning software development in a most productive way. In practice, the project team needs to ensure that team members who are contributing to the system design communicate well with team members who are contributing to the system development throughout the project development cycle. This is in order to streamline the process in such a way as to produce an evolving
design specification that is just enough to guide development of high-quality software.

Second, iterative software releases and the release of a prototype early in the development iteration are recommended in order to solicit feedback from end users. The software engineering practice of iterative software releases brings end users into the loop in such a way that their feedback can be integrated into the iterative software design and development process as early as possible, thereby ensuring the delivery of a software product with a large user base. It would be regrettable for any software project, especially large-scale, complex scientific projects that require several years of team development effort, to yield an end product with very few end users. The integration of end user feedback throughout the software development cycle via iterative software releases can prevent such a regrettable scenario from happening by addressing end user concerns in a timely manner. Early in the development process, the focus should be on simple designs that best fit the daily workflow of end users in order to ensure efficient delivery of an easy-to-use, high-quality end product.

Last, the adoption of software engineering practices is crucial to ensure software quality and sustainability, but these practices should be applied selectively to individual projects, so as not to hinder research productivity. Through the HydroShare experience three software engineering practices have been identified that warrant particular consideration for making software engineering more feasible and easier to integrate into one’s research activities; namely, code refactoring, code review, and software testing. Code refactoring is needed on occasion in order to make changes to the underlying data structures and frameworks so that subsequent software development will be based on a better foundation, thereby resulting in improvements in software quality and development productivity. Because code refactoring can be very disruptive and may require a great deal of effort, careful consideration must be paid to the costs-benefits before adopting code refactoring. In certain circumstances, proof-of-concept prototyping will be needed in advance of any decision to adopt code refactoring in order to prove that the benefits outweigh the costs. While scientists often assume that output errors are the result of faulty theory rather than faulty software [354], the adoption of code review and software testing as precepts of sound software engineering in large-scale, scientific software development projects will help to minimize output errors and ensure that the final software product is high quality and sustainable.

\section{10.6 Conclusion}

The HydroShare project is a work in progress, and exploration, refinement, and implementation of the topics herein are by no means finished. Rather, the
goal is to provide readers with insight into the HydroShare experience and lessons learned in order to minimize the learning curve and accelerate the development progress for other teams. The goal of this chapter is to provide readers with a basic understanding of why good software engineering for science is tantamount to the success and sustainability of a scientific research project and why poor software engineering will detract from research time, with more time spent managing poorly written code than actually conducting research. In the long run, good software engineering will foster research and one’s research career by ensuring the validity of research findings, reducing the amount of time needed to maintain and extend code, and improving the ease at which new features can be adopted, thus supporting software reuse and sustainability.

Acknowledgments

Technical editorial and writing support was provided by Karamarie Fecho, Ph.D. This material is based upon work supported by the NSF under awards 1148453 and 1148090; any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. The authors wish to thank many who have contributed to the HydroShare project, including but not limited to: Jennifer Arrigo, Larry Band, Christina Bandaragoda, Alex Bedig, Brian Blanton, Jeff Bradberry, Chris Calloway, Claris Castillo, Tony Castronova, Mike Conway, Jason Coposky, Shawn Crawley, Antoine deTorcey, Tian Gan, Jon Goodall, Ilan Gray, Jeff Heard, Rick Hooper, Harry Johnson, Drew (Zhiyu) Li, Rob Lineberger, Yan Liu, Shaun Livingston, David Maidment, Phyllis Mbewe, Venkatesh Merwade, Stephanie Mills, Mohamed Morsy, Jon Pollak, Mauriel Ramirez, Terrell Russell, Jeff Sadler, Martin Seul, Kevin Smith, Carol Song, Lisa Stillwell, Nathan Swain, Sid Thakur, David Valentine, Tim Whiteaker, Zhaokun Xue, Lan Zhao, and Shandian Zhe.

The authors wish to especially thank Stan Ahalt, Director of RENCI, and Ashok Krishnamurthy, Deputy Director of RENCI for their continued organizational and supplemental financial support of this project.
References


References


References


[61] CRAN. The comprehensive R archive network. [https://cran.r-project.org/](https://cran.r-project.org/), 2014.


References


References


References


References


References


References


References


References


References


258

References


References


References


References


