Software Development, Design and Coding

With Patterns, Debugging, Unit Testing, and Refactoring

—

Learn the principles of good software design, and how to turn those principles into great code

—

Second Edition

—

John F. Dooley

Apress®

Software Development, Design and Coding

With Patterns, Debugging, Unit Testing, and Refactoring

Second Edition

John F. Dooley

Apress®

Software Development, Design and Coding

John F. Dooley
Galesburg, Illinois, USA

ISBN-13 (pbk): 978-1-4842-3152-4 ISBN-13 (electronic): 978-1-4842-3153-1
https://doi.org/10.1007/978-1-4842-3153-1

Library of Congress Control Number: 2017961306

Cover image designed by Vexels.com

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Todd Green
Development Editor: James Markham
Technical Reviewer: Michael Thomas
Coordinating Editor: Jill Balzano
Copy Editor: Corbin Collins
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484231524. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

For Diane, who is always there, and for Patrick,
the best son a guy could have.

Contents

About the Author xv

About the Technical Reviewer xvii

Acknowledgments xix

Preface xxi

Chapter 1: Introduction to Software Development 1

What We're Doing 2

So, How to Develop Software? 2

Conclusion 4

References 5

Chapter 2: Software Process Models 7

The Four Variables 8

A Model That's not a Model At All: Code and Fix 8

Cruising over the Waterfall 9

Iterative Models 11

Evolving the Iterative Model 12

Risk: The Problem with Plan-Driven Models 13

Agile Methodologies 13

Agile Values and Principles 14

eXtreme Programming (XP) 15

XP Overview 15

The Four Basic Activities 16

Implementing XP: The 12 Practices 16

Scrum 18
Scrum Roles 19
The Sprint 19
Scrum Artifacts 19
Sprint Flow 20
Lean Software Development 21
Principle 1: Eliminate Waste 22
Principle 2: Build Quality In 22
Principle 3: Create Knowledge 22
Principle 4: Defer Commitment 23
Principle 5: Deliver Fast 23
Principle 6: Respect People 23
Principle 7: Optimize the Whole 24
Kanban 24
The Kanban board, WIP, and Flow 24
Lead Time 26
Conclusion 26
References 27
Chapter 3: Project Management Essentials 29
Project Planning 30
Project Organization 30
Risk Analysis 31
Resource Requirements 33
Task Estimates 33
Project Schedule 34
Velocity 35
Project Oversight 36
Status Reviews and Presentations 36
Defects 37

The Retrospective 38
Conclusion 38
References 38
Chapter 4: Requirements 39
What Types of Requirements Are We Talking About? 39
User Requirements 40
Domain Requirements 40
Non-Functional Requirements 40
Non-Requirements 41
Requirements Gathering in a Plan-Driven Project 41
But I Don't Like Writing! 41
Outline of a Functional Specification 42
Design and New Feature Ideas 43
One More Thing 44
Requirements Gathering in an Agile Project 44
The Three Cs 44
INVEST in Stories 45
Product Backlog 47
SMART Tasks 47
Sprint/Iteration Backlog 48
Requirements Digging 48
Why Requirements Digging Is Hard 49
Analyzing the Requirements 50
Conclusion 51
References 51
Chapter 5: Software Architecture 53
General Architectural Patterns 54
The Main Program—Subroutine Architectural Pattern 54
Pipe-and-Filter Architecture 55

An Object-Oriented Architectural Pattern 56
An MVC Example: Let's Hunt! 58
The Client-Server Architectural Pattern 60
The Layered Approach 61
Conclusion 62
References 63
Chapter 6: Design Principles 65
The Design Process 68
Desirable Design Characteristics (Things Your Design Should Favor) 69
Design Heuristics 70
Designers and Creativity 72
Conclusion 73
References 74
Chapter 7: Structured Design 75
Structured Programming 75
Stepwise Refinement 76
Example of Stepwise Refinement: The Eight-Queens Problem 77
Modular Decomposition 84
Example: Keyword in Context 86
Conclusion 94
References 94
Chapter 8: Object-Oriented Overview 95
An Object-Oriented Analysis and Design Process 96
Requirements Gathering and Analysis 98
Design 98
Implementation and Testing 98
Release/Maintenance/Evolution 98

Doing the Process 98
The Problem Statement 98
The Feature List 99
Use Cases 99
Decompose the Problem 100
Class Diagrams 100
Code Anyone? 101
Conclusion 105
References 106
Chapter 9: Object-Oriented Analysis and Design 107
Analysis 108
An Analytical Example 109
Design 111
Change in the Right Direction 112
Recognizing Change 112
Songbirds Forever 113
A New Requirement 113
Separating Analysis and Design 115
Shaping the Design 116
Abstraction 117
Conclusion 119
References 120
Chapter 10: Object-Oriented Design Principles 121
List of Fundamental Object-Oriented Design Principles 122
Encapsulate Things in Your Design That Are Likely to Change 122
Code to an Interface Rather Than to an Implementation 123
The Open-Closed Principle 126
The Don't Repeat Yourself Principle 127
The Single Responsibility Principle 128

The Liskov Substitution Principle 129
The Dependency Inversion Principle 136
The Interface Segregation Principle 138
The Principle of Least Knowledge 138
Class Design Guidelines 139
Conclusion 140
References 140

Chapter 11: Design Patterns 141

Design Patterns and the Gang of Four 142
The Classic Design Patterns 143
Patterns We Can Use 144
Creational Patterns 144
Structural Patterns 151
Behavioral Patterns 157
Conclusion 166
References 166

Chapter 12: Parallel Programming 167

Concurrency vs. Parallelism 168
Parallel Computers 170
Flynn's Taxonomy 170
Parallel Programming 171
Scalability 172
Performance 172
Obstacles to Performance Improvement 173
How to Write a Parallel Program 174
Parallel Programming Models 174
Designing Parallel Programs 175
Parallel Design Techniques 175

Programming Languages and APIs (with examples) 177
Parallel Language Features 177
Java Threads 178
OpenMP 184
The Last Word on Parallel Programming 188
References 189
Chapter 13: Parallel Design Patterns 191
Parallel Patterns Overview 191
Parallel Design Pattern Design Spaces 192
A List of Parallel Patterns 199
Embarrassingly Parallel 199
Master/Worker 200
Map and Reduce 200
MapReduce 202
Divide & Conquer 204
Fork/Join 205
A Last Word on Parallel Design Patterns 209
References 209
Chapter 14: Code Construction 211
A Coding Example 213
Functions and Methods and Size 214
Formatting, Layout, and Style 214
General Layout Issues and Techniques 215
White Space 217
Block and Statement Style Guidelines 217
Declaration Style Guidelines 218
Commenting Style Guidelines 220
Identifier Naming Conventions 222

Refactoring 224
When to Refactor 224
Types of Refactoring 226
Defensive Programming 228
Assertions Are Helpful 229
Exceptions 230
Error Handling 230
Exceptions in Java 232
The Last Word on Coding 234
References 234
Chapter 15: Debugging 235
What Is an Error, Anyway? 236
What Not To Do 237
An Approach to Debugging 238
Reproduce the Problem Reliably 238
Find the Source of the Error 239
Fix the Error (Just That One)! 245
Test the Fix 246
Look for More Errors 246
Source Code Control 246
The Collision Problem 247
Source Code Control Systems 248
One Last Thought on Coding and Debugging: Pair Programming 250
Conclusion 251
References 251
Chapter 16: Unit Testing 253
The Problem with Testing 254
That Testing Mindset 254
When to Test? 255

Testing in an Agile Development Environment 256
What to Test? 256
Code Coverage: Test Every Statement 257
Data Coverage: Bad Data Is Your Friend? 258
Characteristics of Tests 259
How to Write a Test 259
The Story 260
The Tasks 260
The Tests 260
JUnit: A Testing Framework 264
Testing Is Good 268
Conclusion 268
References 269
Chapter 17: Code Reviews and Inspections 271
Walkthroughs, Reviews, and Inspections 272
Walkthroughs 273
Code Reviews 273
Code Inspections 274
Inspection Roles 275
Inspection Phases and Procedures 276
Reviews in Agile Projects 278
How to Do an Agile Peer Code Review 279
Summary of Review Methodologies 279
Defect Tracking Systems 280
Defect Tracking in Agile Projects 281
Conclusion 282
References 282

Chapter 18: Ethics and Professional Practice 283

Introduction to Ethics 283

Ethical Theory 284

Deontological Theories 284

Consequentialism (Teleological Theories) 287

Ethical Drivers 289

Legal Drivers 289

Professional Drivers 289

Ethical Discussion and Decision Making 291

Identifying and Describing the Problem 291

Analyzing the Problem 291

Case Studies 292

#1 Copying Software 292

#2 Who's Computer Is It? 292

#3 How Much Testing Is Enough? 292

#4 How Much Should You Tell? 293

The Last Word on Ethics? 293

References 294

The ACM Code of Ethics and Professional Conduct 294

Preamble 294

Contents & Guidelines 295

The ACM/IEEE-CS Software Engineering Code of Ethics 300

PREAMBLE 300

PRINCIPLES 301

Chapter 19: Wrapping It all Up 305

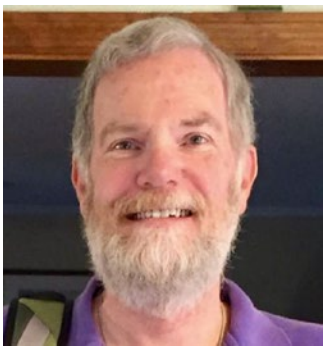
What Have You Learned? 305

What to Do Next? 306

References 308

Index 311

About the Author



John F. Dooley is the William and Marilyn Ingersoll Emeritus Professor of Computer Science at Knox College in Galesburg, Illinois. Before returning to teaching in 2001, Professor Dooley spent nearly 18 years in the software industry as a developer, designer, and manager working for companies such as Bell Telephone Laboratories, McDonnell Douglas, IBM, and Motorola, along with an obligatory stint as head of development at a software startup. He has more than two dozen professional journal and conference publications and four books to his credit, along with numerous presentations. He has been a reviewer for the Association for Computing Machinery Special Interest Group on Computer Science Education (SIGCSE) Technical Symposium for the last 36 years and reviews papers for the IEEE Transactions on Education, the ACM Innovation and Technology in Computer Science Education (ITiCSE) Conference, and other professional conferences. He has developed short courses in software development and created three separate software engineering courses at the advanced undergraduate level.

About the Technical Reviewer

Michael Thomas has worked in software development for more than 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has more than ten years experience working with mobile devices. His current focus is in the medical sector using mobile devices to accelerate information transfer between patients and healthcare providers.

Acknowledgments

I'd like to thank Todd Green of Apress for encouraging me and making this book possible. The staff at Apress, especially Jill Balzano and Michael Thomas, have been very helpful and gracious. The book is much better for their reviews, comments, and edits.

Thanks also to all my students in CS 292 over the last 12 years, who have put up with successive versions of the course notes that became this book, and to my CS department colleagues David Bunde and Jaime Spacco, who put up with me for all these years. And my thanks also go to Knox College for giving me the time and resources to finish both editions of this book.

Finally, I owe everything to Diane, who hates that I work nights, but loves that I can work at home.

Preface

What's this book all about? Well, it's about how to develop software from a personal perspective. We'll look at what it means for you to take a problem and produce a program to solve it from beginning to end. That said, this book focuses a lot on design. How do you design software? What things do you take into account? What makes a good design? What methods and processes are there to help you design software? Is designing small programs different from designing large ones? How can you tell a good design from a bad one? What general patterns can you use to help make your design more readable and understandable?

It's also about code construction. How do you write programs and make them work? "What?" you say. "I've already written eight gazillion programs! Of course I know how to write code!" Well, in this book, we'll explore what you already do and investigate ways to improve on that. We'll spend some time on coding standards, debugging, unit testing, modularity, and characteristics of good programs. We'll also talk about reading code, what makes a program readable, and how to review code that others have written with an eye to making it better. Can good, readable code replace documentation? How much documentation do you really need?

And it's about software engineering, which is usually defined as "the application of engineering principles to the development of software." What are *engineering principles*? Well, first, all engineering efforts follow a defined *process*. So we'll be spending a bit of time talking about how you run a software development project and what phases there are to a project. We'll talk a lot about agile methodologies, how they apply to small development teams and how their project management techniques work for small- to medium-sized projects. All engineering work has a basis in the application of science and mathematics to real-world problems. So does software development. As I've said already, we'll be spending *a lot* of time examining how to design and implement programs that solve specific problems.

By the way, there's at least one other person (besides me) who thinks software development is not an engineering discipline. I'm referring to Alistair Cockburn, and you can read his paper, "The End of Software Engineering and the Start of Economic-Cooperative Gaming," at http://alistair.cockburn.us/The+end+of+software+engineering+and+the+start+of+economic-cooperative+gaming.

Finally, this book is about professional practice, the ethics and the responsibilities of being a software developer, social issues, privacy, how to write secure and robust code, and the like. In short, those fuzzy other things that one needs in order to be a *professional* software developer.

This book covers many of the topics described for the ACM/IEEE Computer Society *Curriculum Guidelines for Undergraduate Degree Programs in Computer Science* (known as CS2013).[1] In particular, it covers topics in a number of the Knowledge Areas of the Guidelines, including Software Development Fundamentals, Software Engineering, Systems Fundamentals, Parallel and Distributed Computing, Programming Languages, and Social Issues and Professional Practice. It's designed to be both a textbook

[1]The Joint Task Force on Computing Education. 2013. "Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science." New York, NY: ACM/IEEE Computer Society. www.acm.org/education/CS2013-final-report.pdf.

for a junior-level undergraduate course in software design and development and a manual for the working professional. Although the chapter order generally follows the standard software development sequence, one can read the chapters independently and out of order. I'm assuming that you already know how to program and that you're conversant with at least one of these languages: Java, C, or C++. I'm also assuming you're familiar with basic data structures, including lists, queues, stacks, maps, and trees, along with the algorithms to manipulate them.

In this second edition, most of the chapters have been updated, some new examples have been added, and the book discusses modern software development processes and techniques. Much of the plan-driven process and project-management discussions from the first edition have been removed or shortened, and longer and new discussions of agile methodologies, including Scrum, Lean Software Development, and Kanban have taken their place. There are new chapters on parallel programming and parallel design patterns, and a new chapter on ethics and professional practice.

I use this book in a junior-level course in software development. It's grown out of the notes I've developed for that class over the past 12 years. I developed my own notes because I couldn't find a book that covered all the topics I thought were necessary for a course in software development, as opposed to one in software engineering. Software engineering books tend to focus more on process and project management than on design and actual development. I wanted to focus on the design and writing of real code rather than on how to run a large project. Before beginning to teach, I spent nearly 18 years in the computer industry, working for large and small companies, writing software, and managing other people who wrote software. This book is my perspective on what it takes to be a software developer on a small- to medium-sized team and help develop great software.

I hope that by the end of the book you'll have a much better idea of what the design of good programs is like, what makes an effective and productive developer, and how to develop larger pieces of software. You'll know a lot more about design issues. You'll have thought about working in a team to deliver a product to a written schedule. You'll begin to understand project management, know some metrics and how to review work products, and understand configuration management. I'll not cover everything in software development—not by a long stretch—and we'll only be giving a cursory look at the management side of software engineering, but you'll be in a much better position to visualize, design, implement, and test software of many sizes, either by yourself or in a team.

CHAPTER 1

Introduction to Software Development

"Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any—no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware. We cannot expect ever to see twofold gains every two years."

— Frederick J. Brooks, Jr.[1]

So, you're asking yourself, why is this book called *Software Development, Design and Coding*? Why isn't it called *All About Programming* or *Software Engineering*? After all, isn't that what software development is? Well, no. Programming is a part of software development, but it's certainly not all of it. Likewise, software development is a part of software engineering, but it's not all of it.

Here's the definition of software development that we'll use in this book: *software development* is the process of taking a set of requirements from a user (a problem statement), analyzing them, designing a solution to the problem, and then implementing that solution on a computer.

Isn't that programming, you ask? No. *Programming* is really the implementation part, or possibly the design and implementation part, of software development. Programming is central to software development, but it's not the whole thing.

Well, then, isn't it software engineering? Again, no. *Software engineering* also involves a process and includes software development, but it also includes the entire management side of creating a computer program that people will use, including project management, configuration management, scheduling and estimation, baseline building and scheduling, managing people, and several other things. Software development is the fun part of software engineering.

So, software development is a narrowing of the focus of software engineering to just that part concerned with the creation of the actual software. And it's a broadening of the focus of programming to include analysis, design, and release issues.

[1]Brooks, Frederick. "No Silver Bullet." *IEEE Computer* (1987). 20(4): 10-19.

J. F. Dooley, *Software Development, Design and Coding*, https://doi.org/10.1007/978-1-4842-3153-1_1

What We're Doing

It turns out that, after 70 or so years of using computers, we've discovered that developing software is hard. Learning how to develop software correctly, efficiently, and beautifully is also hard. You're not born knowing how to do it, and many people, even those who take programming courses and work in the industry for years, don't do it particularly well. It's a skill you need to pick up and practice—a lot. You don't learn programming and development by reading books—not even this one. You learn it by doing it. That, of course, is the attraction: to work on interesting and difficult problems. The challenge is to work on something you've never done before, something you might not even know if you can solve. That's what has you coming back to create new programs again and again.

There are probably several ways to learn software development. But I think that all of them involve reading excellent designs, reading a lot of code, writing a lot of code, and thinking deeply about how you approach a problem and design a solution for it. Reading a lot of code, especially really beautiful and efficient code, gives you lots of good examples about how to think about problems and approach their solution in a particular style. Writing a lot of code lets you experiment with the styles and examples you've seen in your reading. Thinking deeply about problem solving lets you examine how you work and how you do design, and lets you extract from your labors those patterns that work for you; it makes your programming more intentional.

So, How to Develop Software?

The first thing you should do is read this book. It certainly won't tell you everything, but it will give you a good introduction into what software development is all about and what you need to do to write great code. It has its own perspective, but that's a perspective based on 20 years writing code professionally and another 22 years trying to figure out how to teach others to do it.

Despite the fact that software development is only part of software engineering, software development is the heart of every software project. After all, at the end of the day what you deliver to the user is working code. A team of developers working in concert usually creates that code. So, to start, maybe we should look at a software project from the outside and ask what does that team need to do to make that project a success?

In order to do software development well, you need the following:

- *A small, well-integrated team*: Small teams have fewer lines of communication than larger ones. It's easier to get to know your teammates on a small team. You can get to know their strengths and weaknesses, who knows what, and who is the "go-to" person for particular problems or particular tools. Well-integrated teams have usually worked on several projects together. Keeping a team together across several projects is a major job of the team's manager. Well-integrated teams are more productive, are better at holding to a schedule, and produce code with fewer defects at release. The key to keeping a team together is to give them interesting work to do and then leave them alone.

- *Good communication among team members*: Constant communication among team members is critical to day-to-day progress and successful project completion. Teams that are co-located are better at communicating and communicate more than teams that are distributed geographically (even if they're just on different floors or wings of a building). This is a major issue with larger companies that have software development sites scattered across the globe.

- *Good communication between the team and the customer*: Communication with the customer is essential to controlling requirements and requirements churn during a project. On-site or close-by customers allow for constant interaction with the development team. Customers can give immediate feedback on new releases and be

involved in creating system and acceptance tests for the product. Agile development methodologies strongly encourage customers to be part of the development team and, even better, to be on site daily. See Chapter 2 for a quick introduction to a couple of agile methodologies.

- *A process that everyone buys into*: Every project, no matter how big or small, follows a process. Larger projects and larger teams tend to be more plan-driven and follow processes with more rules and documentation required. Larger projects require more coordination and tighter controls on communication and configuration management. Smaller projects and smaller teams will, these days, tend to follow more agile development processes, with more flexibility and less documentation required. This certainly doesn't mean there is *no* process in an agile project; it just means you do what makes sense for the project you're writing so that you can satisfy all the requirements, meet the schedule, and produce a quality product. See Chapter 2 for more details on process and software life cycles.

- *The ability to be flexible about that process*: No project ever proceeds as you think it will on the first day. Requirements change, people come and go, tools don't work out or get updated, and so on. This point is all about handling risk in your project. If you identify risks, plan to mitigate them, and then have a contingency plan to address the event where the risk actually occurs, you'll be in much better shape. Chapter 4 talks about requirements and risk.

- *A plan that every one buys into*: You wouldn't write a sorting program without an algorithm to start with, so you shouldn't launch a software development project without a plan. The *project plan* encapsulates what you're going to do to implement your project. It talks about process, risks, resources, tools, requirements management, estimates, schedules, configuration management, and delivery. It doesn't have to be long and it doesn't need to contain all the minute details of the everyday life of the project, but everyone on the team needs to have input into it, they need to understand it, and they need to agree with it. Unless everyone buys into the plan, you're doomed. See Chapter 3 for more details on project plans.

- *To know where you are at all times*: It's that communication thing again. Most projects have regular status meetings so that the developers can "sync up" on their current status and get a feel for the status of the entire project. This works very well for smaller teams (say, up to about 20 developers). Many small teams will have daily meetings to sync up at the beginning of each day. Different process models handle this "stand-up" meeting differently. Many plan-driven models don't require these meetings, depending on the team managers to communicate with each other. Agile processes often require daily meetings to improve communications among team members and to create a sense of camaraderie within the team.

- *To be brave enough to say, "Hey, we're behind!"*: Nearly all software projects have schedules that are too optimistic at the start. It's just the way we developers are. Software developers are an optimistic bunch, generally, and it shows in their estimates of work. "Sure, I can get that done in a week!" "I'll have it to you by the end of the day." "Tomorrow? Not a problem." No, no, no, no, no. Just face it. At some point you'll be behind. And the best thing to do about it is tell your manager right away. Sure, she might be angry—but she'll be angrier when you end up a month behind and she didn't know it. Fred Brooks's famous answer to the question of how software projects get so far behind is "one day at a time." The good news, though, is that the earlier you figure out you're behind, the more options you have. These include lengthening the schedule (unlikely, but it does happen), moving some requirements to a future release, getting additional help, and so on. The important part is to keep your manager informed.

- *The right tools and the right practices for this project*: One of the best things about software development is that every project is different. Even if you're doing version 8.0 of an existing product, things change. One implication of this is that for every project, one needs to examine and pick the right set of development tools for this particular project. Picking tools that are inappropriate is like trying to hammer nails with a screwdriver; you might be able to do it eventually, but is sure isn't easy or pretty or fun, and you can drive a lot more nails in a shorter period of time with a hammer. The three most important factors in choosing tools are the application type you are writing, the target platform, and the development platform. You usually can't do anything about any of these three things, so once you know what they are, you can pick tools that improve your productivity. A fourth and nearly as important factor in tool choice is the composition and experience of the development team. If your team is composed of all experienced developers with facility on multiple platforms, tool choice is pretty easy. If, on the other hand, you have a bunch of fresh-outs and your target platform is new to all of you, you'll need to be careful about tool choice and fold in time for training and practice with the new tools.
- *To realize that you don't know everything you need to know at the beginning of the project*: Software development projects just don't work this way. You'll always uncover new requirements. Other requirements will be discovered to be not nearly as important as the customer thought, and still others that were targeted for the next release are all of a sudden requirement number 1. Managing *requirements churn* during a project is one of the single most important skills a software developer can have. If you're using new development tools—say, that new web development framework—you'll uncover limitations you weren't aware of and side-effects that cause you to have to learn, for example, three other tools to understand them—for example, that that web development tool is Ruby based, requires a specific relational database system to run, and needs a particular configuration of Apache to work correctly.

Conclusion

Software development is the heart of every software project and is the heart of software engineering. Its objective is to deliver excellent, defect-free code to users on time and within budget—all in the face of constantly changing requirements. That makes development a particularly hard job to do. But finding a solution to a difficult problem and getting your code to work correctly is just about the coolest feeling in the world.

> *"[Programming is] the only job I can think of where I get to be both an engineer and an artist. There's an incredible, rigorous, technical element to it, which I like because you have to do very precise thinking. On the other hand, it has a wildly creative side where the boundaries of imagination are the only real limitation. The marriage of those two elements is what makes programming unique. You get to be both an artist and a scientist. I like that. I love creating the magic trick at the center that is the real foundation for writing the program. Seeing that magic trick, that essence of your program, working correctly for the first time, is the most thrilling part of writing a program."*
>
> —Andy Hertzfeld (designer of the first Mac OS)[2]

[2]Lammers, Susan. *Programmers at Work*. (Redmond, WA: Microsoft Press, 1986).

References

Brooks, Frederick. “No Silver Bullet.” *IEEE Computer* (1987). 20(4): 10-19.
Lammers, Susan. *Programmers at Work*. (Redmond, WA: Microsoft Press, 1986).

CHAPTER 2

Software Process Models

If you don't know where you're going, any road will do.

If you don't know where you are, a map won't help.

—Watts Humphrey

Every program has a life cycle. It doesn't matter how large or small the program is, or how many people are working on the project—all programs go through the same steps:

1. Conception
2. Requirements gathering/exploration/modeling
3. Design
4. Coding and debugging
5. Testing
6. Release
7. Maintenance/software evolution
8. Retirement

Your program may compress some of these steps, or combine two or more steps into a single piece of work, but all programs go through all steps of the life cycle.

Although every program has a life cycle, many different process variations encompass these steps. Every development process, however, is a variation on two fundamental types. In the first type, the project team will generally do a complete life cycle—at least steps 2 through 7—before they go back and start on the next version of the product. In the second type, which is more prevalent now, the project team will generally do a partial life cycle—usually steps 3 through 5—and iterate through those steps several times before proceeding to the release step.

These two general process types can be implemented using two classes of project management models. These are traditional *plan-driven models*,[1] and the newer *agile development* models.[2] In plan-driven models, the methodology tends to be stricter in terms of process steps and when releases happen. Plan-driven models have more clearly defined phases, and more requirements for sign-off on completion of a phase

[1]Paulk, M. C. *The Capability Maturity Model: Guidelines for Improving the Software Process*. (Reading, MA: Addison-Wesley, 1995.)
[2]Martin, R. C. *Agile Software Development, Principles, Patterns, and Practices*. (Upper Saddle River, NJ: Prentice Hall, 2003.)

J. F. Dooley, *Software Development, Design and Coding*, https://doi.org/10.1007/978-1-4842-3153-1_2

before moving on to the next phase. Plan-driven models require more documentation at each phase and verification of completion of each work product. These tend to work well for large contracts for new software with well-defined deliverables. The agile models are inherently incremental and make the assumption that small, frequent releases produce a more robust product than larger, less frequent ones. Phases in agile models tend to blur together more than in plan-driven models, and there tends to be less documentation of work products required, the basic idea being that code is what is being produced, so developer efforts should focus there. See the Agile Manifesto web page at http://agilemanifesto.org to get a good feel for the agile development model and goals.

This chapter takes a look at several software life cycle models, both plan driven and agile, and compares them. There is no one best process for developing software. Each project must decide on the model that works best for its particular application and base that decision on the project domain, the size of the project, the experience of the team, and the timeline of the project. But first we have to look at the four factors, or variables, that all software development projects have in common.

The Four Variables

The four variables of software development projects are as follows:

- *Cost* is probably the most constrained; you can't spend your way to quality or being on schedule, and as a developer you have very limited control over cost. Cost can influence the size of the team or, less often, the types of tools available to the team. For small companies and startups, cost also influences the environment where the developers will work.
- *Time* is your delivery schedule and is unfortunately many times imposed on you from the outside. For example, most consumer products (be they hardware or software) will have a delivery date somewhere between August and October in order to hit the holiday buying season. You can't move Christmas. If you're late, the only way to fix your problem is to drop features or lessen quality, neither of which is pretty. Time is also where Brooks's law gets invoked (adding programmers to a late project just makes it later).
- *Quality* is the number and severity of defects you're willing to release with. You can make short-term gains in delivery schedules by sacrificing quality, but the cost is enormous: it will take more time to fix the next release, and your credibility is pretty well shot.
- *Features* (also called *scope*) are what the product actually does. This is what developers should always focus on. It's the most important of the variables from the customer's perspective and is also the one you as a developer have the most control over. Controlling scope allows you to provide managers and customers control over quality, time, and cost. If the developers don't have control over the feature set for each release, then they are likely to blow the schedule. This is why developers should do the estimates for software work products.

A Model That's not a Model At All: Code and Fix

The first model of software development we'll talk about isn't really a model at all. But it is what most of us do when we're working on small projects by ourselves, or maybe with a single partner. It's the *code and fix model*.

The code and fix model, shown in Figure 2-1, is often used in lieu of actual project management. In this model there are no formal requirements, no required documentation, and no quality assurance or formal testing, and release is haphazard at best. Don't even think about effort estimates or schedules when using this model.

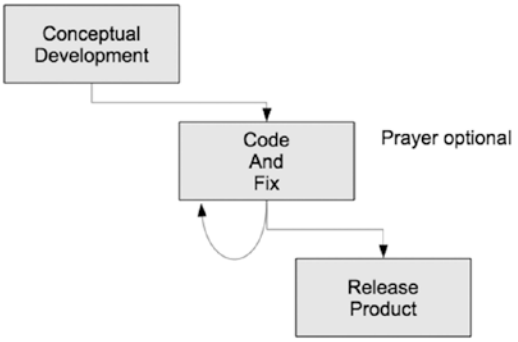


Figure 2-1. *The code and fix process model*

Code and fix says take a minimal amount of time to understand the problem and then start coding. Compile your code and try it out. If it doesn't work, fix the first problem you see and try it again. Continue this cycle of type-compile-run-fix until the program does what you want with no fatal errors and then ship it.

Every programmer knows this model. We've all used it way more than once, and it actually works in certain circumstances: for quick, disposable tasks. For example, it works well for proof-of-concept programs. There's no maintenance involved, and the model works well for small, single-person programs. It is, however, a *very dangerous* model for any other kind of program.

With no real mention of configuration management, little in the way of testing, no architectural planning, and probably little more than a desk check of the program for a code review, this model is good for quick and dirty prototypes and really nothing more. Software created using this model will be small, short on user interface niceties, and idiosyncratic.

That said, code and fix is a terrific way to do quick and dirty prototypes and short, one-off programs. It's useful to validate architectural decisions and to show a quick version of a user interface design. Use it to understand the larger problem you're working on.

Cruising over the Waterfall

The first and most traditional of the plan-driven process models is the *waterfall* model. Illustrated in Figure 2-2, it was created in 1970 by Winston Royce,[3] and addresses all of the standard life cycle phases. It progresses nicely through requirements gathering and analysis, to architectural design, detailed design, coding, debugging, integration and system testing, release, and maintenance. It requires detailed documentation at each stage, along with reviews, archiving of the documents, sign-offs at each process phase, configuration management, and close management of the entire project. It's an exemplar of the plan-driven process.

[3]Royce, W. W. *Managing the Development of Large Software Systems*. Proceedings of IEEE WESCON. (Piscataway, NJ, IEEE Press. 1970.)

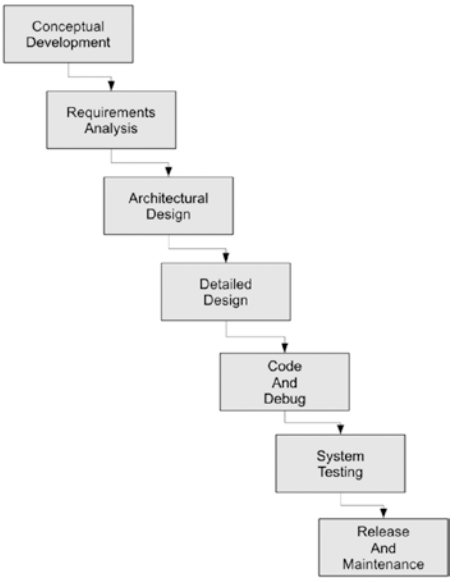


Figure 2-2. *The waterfall process model*

It also doesn't work.

There are two fundamental and related problems with the waterfall model that hamper its acceptance and make it very difficult to implement. First, it generally requires that you finish phase N before you continue on to phase N+1. In the simplest example, this means you must nail down *all* your requirements before you start your architectural design, and finish your coding and debugging before you start anything but unit testing. In theory, this is great. You'll have a complete set of requirements, you'll understand exactly what the customer wants and everything the customer wants, so you can then confidently move on to designing the system.

In practice, though, this never happens. I've never worked on a project where all the requirements were nailed down at the beginning of the work. I've never seen a project where big things didn't change somewhere during development. So, finishing one phase before the other begins is problematic.

The second problem with the waterfall is that, as stated, it has no provision for backing up. It is fundamentally based on an assembly-line mentality for developing software. The nice little diagram shows no way to go back and rework your design if you find a problem during implementation. This is similar to the first problem above. The implications are that you really have to nail down one phase and review everything in detail before you move on. In practice this is just not practical. The world doesn't work this way. You never know everything you need to know at exactly the time you need to know it. This is why software is a wicked problem. Most organizations that implement the waterfall model modify it to have the ability to back up one or more phases so that missed requirements or bad design decisions can be fixed. This helps and generally makes the waterfall model usable, but the requirement to update all the involved documentation when you do back up makes even this version problematic.

All this being said, the waterfall is a terrific theoretical model. It isolates the different phases of the life cycle and forces you to think about what you really do need to know before you move on. It's also a good way to start thinking about very large projects; it gives managers a warm fuzzy because it lets them think they know what's going on (they don't, but that's another story). It's also a good model for inexperienced teams working on a well-defined, new project because it leads them through the life cycle.

Iterative Models

> *The best practice is to iterate and deliver incrementally, treating each iteration as a closed-end "mini-project," including complete requirements, design, coding, integration, testing, and internal delivery. On the iteration deadline, deliver the (fully-tested, fully-integrated) system thus far to internal stakeholders. Solicit their feedback on that work, and fold that feedback into the plan for the next iteration.*
>
> (From "How Agile Projects Succeed"[4])

Although the waterfall model is a great theoretical model, it fails to recognize that all the requirements aren't typically known in advance, and that mistakes will be made in architectural design, detailed design, and coding. Iterative process models make this required change in process steps more explicit and create process models that build products a piece at a time.

In most iterative process models, you'll take the known requirements—a snapshot of the requirements at some time early in the process—and prioritize them, typically based on the customer's ranking of what features are most important to deliver first. Notice also that this is the first time we've got the customer involved except at the beginning of the whole development cycle.

You then pick the highest priority requirements and plan a series of iterations, where each iteration is a complete project. For each iteration, you'll add a set of the next highest priority requirements (including some you or the customer may have discovered during the previous iteration) and repeat the project.
By doing a complete project with a subset of the requirements every time at the end of each iteration, you end up with a complete, working, and robust product, albeit with fewer features than the final product will have.

According to Tom DeMarco, these iterative processes follow one basic rule:

> *Your project, the whole project, has a binary deliverable. On the scheduled completion day, the project has either delivered a system that is accepted by the user, or it hasn't. Everyone knows the result on that day. The object of building a project model is to divide the project into component pieces, each of which has this same characteristic: each activity must be defined by a deliverable with objective completion criteria. The deliverables are demonstrably done or not done."* [5]

So, what happens if you estimate wrong? What if you decide to include too many new features in an iteration? What if there are unexpected delays?

Well, if it looks as if you won't make your iteration deadline, there are only two realistic alternatives: move the deadline or remove features. We'll come back to this problem later when we talk about estimation and scheduling.

The key to iterative development is "live a balanced life—learn some and think some and draw and paint and sing and dance and play and work every day some,"[6] or in the software development world, *analyze* some and *design* some and *code* some and *test* some every day. We'll revisit this idea when we talk about the agile development models later in this chapter.

---

[4] www.adaptionsoft.com/on_time.html
[5] DeMarco, T. *Controlling Software Projects: Management, Measurement and Estimation*. (Upper Saddle River, NJ: Yourdon Press, 1983.)
[6] Fulghum, Robert. *All I Really Need to Know I Learned in Kindergarten*. (New York, NY: Ivy Books. 1986.)

Evolving the Iterative Model

A traditional way of implementing the iterative model is known as evolutionary prototyping.[7] In *evolutionary prototyping*, illustrated in Figure 2-3, one prioritizes requirements as they are received and produces a succession of increasingly feature-rich versions of the product. Each version is refined using customer feedback and the results of integration and system testing. This is an excellent model for an environment of changing or ambiguous requirements, or a poorly understood application domain. This is the model that evolved into the modern agile development processes.

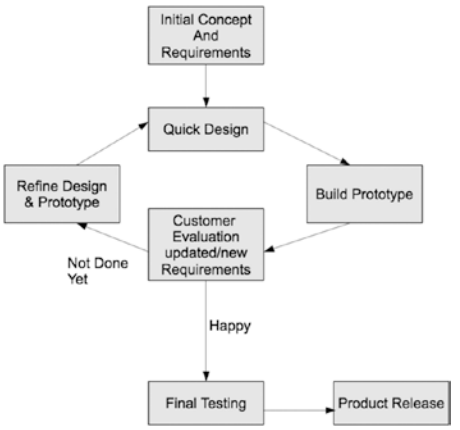


Figure 2-3. *Evolutionary prototyping process model*

Evolutionary prototyping recognizes that it's very hard to plan the full project from the start and that feedback is a critical element of good analysis and design. It's somewhat risky from a scheduling point of view, but when compared to any variation of the waterfall model, it has a very good track record. Evolutionary prototyping provides improved progress visibility for both the customer and project management. It also provides good customer and end user input to product requirements and does a good job of prioritizing those requirements.

On the downside, evolutionary prototyping leads to the danger of unrealistic schedules, budget overruns, and overly optimistic progress expectations. These can happen because the limited number of requirements implemented in a prototype can give the impression of real progress for a small amount of work. On the flip side, putting too many requirements in a single prototype can result is schedule slippages because of overly optimistic estimation. This is a tricky balance to maintain. Because the design evolves over time as the requirements change, there is the possibility of a bad design, unless there's the provision of re-designing—something that becomes harder and harder to do as the project progresses and your customer is more heavily invested in a particular version of the product. There is also the possibility of low maintainability, again because the design and code evolve as requirements change. This may lead to lots of re-work, a broken schedule, and increased difficulty in fixing bugs post-release.

Evolutionary prototyping works best with tight, experienced teams who have worked on several projects together. This type of cohesive team is productive and dexterous, able to focus on each iteration and usually producing the coherent, extensible designs that a series of prototypes requires. This model is not generally recommended for inexperienced teams.

[7]McConnell, S. *Rapid Development: Taming Wild Software Schedules*. (Redmond, WA: Microsoft Press, 1996.)