Software Development, Design, and Coding

With Patterns, Debugging, Unit Testing, and Refactoring

Third Edition

John F. Dooley
Vera A. Kazakova

Apress®

Software Development, Design, and Coding

With Patterns, Debugging, Unit Testing, and Refactoring

Third Edition

John F. Dooley
Vera A. Kazakova

Apress®

Software Development, Design, and Coding: With Patterns, Debugging, Unit Testing, and Refactoring

John F. Dooley
Galesburg, IL, USA

Vera A. Kazakova
Columbia, MD, USA

ISBN-13 (pbk): 979-8-8688-0284-3
ISBN-13 (electronic): 979-8-8688-0285-0
https://doi.org/10.1007/979-8-8688-0285-0

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Melissa Duffy
Development Editor: Laura Berendson
Coordinating Editor: Gryffin Winkler

Cover designed by eStudioCalamar

Cover image by JJ Ying on Unsplash (www.unsplash.com)

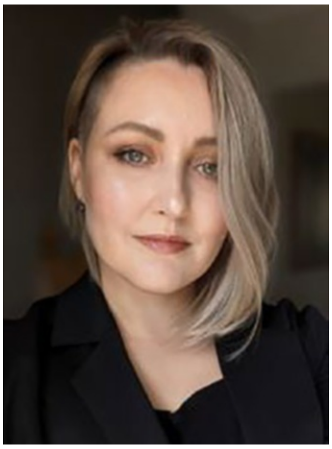
John: For diane once again.

Vera: To all my students.

Table of Contents

About the Authors

John F. Dooley is the William and Marilyn Ingersoll Professor Emeritus of Computer Science at Knox College in Galesburg, Illinois. Before returning to teaching in 2001, Professor Dooley spent more than 16 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 start-up. He has over two dozen professional journal and conference publications and seven 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 he reviews papers for the journal *Cryptologia* and other professional conferences. He has created short courses in software development and three separate software engineering courses at the advanced undergraduate level.



Dr. Vera A. Kazakova is a computer science educator and researcher, with expertise in artificial intelligence, experiential learning, and collaborative methodologies. With a PhD in AI focused on nature-inspired computation and emergent division of labor, her research spans CS education, evolutionary computation, narrative generation, decentralized multiagent systems, and cyber social science. Dr. Kazakova also has extensive experience as a CS educator, having taught programming, artificial intelligence, research, and software development courses. Dr. Kazakova coined the term "Soft-Aware development" to encapsulate a holistic approach for building software, building stakeholder relationships, and building up each developer along the way. An ardent proponent of experiential learning and agile methodologies, Dr. Kazakova champions a multi-sprint learning architecture that enables students to adapt and iterate, fostering a shared environment of continuous growth. Her passion for collaboration, from simplistic autonomous agents to human developers and members of large online communities, sets her apart as an advocate for a more interconnected, empathetic, and empowering approach to CS research, education, and software development.

About the Technical Reviewer

Dr. Takako Soma is an Associate Professor of Computer Science at Illinois College in Jacksonville, Illinois. She is also a co-author of *Guide to Java: A Concise Introduction to Programming* Second Edition (Springer 2023) and *Guide to Data Structures: A Concise Introduction Using Java* (Springer 2017).

Acknowledgments

We'd like to thank Melissa Duffy and Shonmirin P. A. of Apress for making this new edition possible. Our Technical Reviewer and all the staff at Apress have been very helpful and gracious. The book is much better for their reviews, comments, and edits.

Thanks also to all of the students in CS 292 over the years who have used successive versions of this material, first as course notes and then as the finished book, and to our Knox College Computer Science colleagues David Bunde and Jaime Spacco who've listened to us. Finally, thanks to Knox College for giving us the time and resources to finish all the editions of this book.

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!" In this book, we'll explore what you already do and investigate ways to improve on it. 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 talk about what phases there are to this process, as well as how to best support development through becoming an effective facilitator. 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. We will often ground our theoretical discussion by designing and implementing solutions to specific problems.

By the way, there's at least one other person (besides this book's authors) who thinks software development is not an engineering discipline. We're 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, interpersonal skills, privacy, how to write secure and robust code, and the like. In short, those various non-technical things that you need 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 CS2023).[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 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, you can read the chapters independently and out of order. We're assuming that you already know how to program and that you're conversant with at least one of these languages: Java, C, or C++. We are 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 third edition, several chapters have been rewritten and all of the chapters have been updated, including new content and examples. The book discusses modern software development processes and techniques; notably the coverage of agile techniques has been updated and expanded. Much of the plan-driven process and project-management discussions from the second 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 is a new chapter on intellectual property, ownership, and obligations. Finally, the chapter on project management essentials has been greatly expanded to include an introduction and discussion of Soft-Aware development, an approach to software development based on the idea that learning to make software is less crucial than *learning to work together while attempting to make software.*

[1] The Joint Task Force on Computing Education. 2023. "Computer Science Curricula 2023: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science." New York, NY: ACM/IEEE Computer Society. `https://csed.acm.org/wp-content/uploads/2023/03/Version-Beta-v2.pdf`.

We have used this book in an upper-level course in software development and it has grown out of the notes we developed for that class. We developed our own notes because we couldn't find a book that covered all the topics we thought were necessary for a course in software development, as opposed to one in software engineering or just programming. Software engineering books tend to focus more on process and project management than on design and actual development. We wanted to focus on the design and writing of real code rather than on how to run a large project. This book is our perspective on what it takes to be a software developer on a small- to medium-sized team and help develop great software.

We believe 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. We will 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 might be 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.

But isn't that programming? Well, no. Programming is really just 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. Software

[1] Brooks, F. 1987. "No Silver Bullet." *IEEE Computer* 20 (4): 10–19. `www.inst.eecs.berkeley.edu/~maratb/readings/NoSilverBullet.html`.

engineering includes 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.

What We're Doing

It turns out that, after 80 or so years of using computers, we've discovered that developing software is hard. Learning how to develop software effectively, efficiently, and sustainably 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 developing software. 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 we think that all of them involve reading excellent designs, reading a lot of code, writing a lot of code, and thinking deeply about how to 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?

Well, 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 it's a perspective based on our combined 40 years or so of writing code professionally and another 24 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 succeed at software development, you need the following:

> *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; still others that were targeted for the next release are all of a sudden requirement number one. This is known as *churn*. Managing requirements churn during a project is one of the single most important skills a software developer can have. If you are using new development tools (say a 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 (e.g., that web development tool you want to use is Ruby-based, requires a specific relational database system to run, and needs a particular configuration of Apache to work correctly.)
>
> *A small, well integrated team*. Small teams have fewer lines of communication than larger ones. It's easier to get to know your teammates' strengths and weaknesses, understand their personalities and preferences, and establish who is the go-to person for particular problems or 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 and of the individual teammates themselves. Well-integrated teams are more productive, better at holding to a schedule, and more likely to produce code with fewer defects at release. The key to keeping a team together is to give them interesting work, give them the freedom to decide how to do the work, and facilitate the process by removing barriers and helping with conflict resolution.

Good communication among team members. Continuous direct communication among team members is critical to day-to-day progress and successful project completion. Teams that are co-located are generally 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) or that are working virtually.[2] 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 can 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 some agile methodologies.

A process that everyone buys into. Every project, no matter how big or small, follows a process. Larger projects require more coordination and tighter controls on communication and configuration management. As a result, larger teams tend to be more plan-driven and follow processes with more rules and documentation required. 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 current stage of your project, so that you can correctly uncover and satisfy all the requirements, meet the schedule, and produce a quality product. See Chapter 2 for more details on process and software life cycles.

[2] Note that teams that are distributed geographically can also be closer to clients and thus have better communication with them. Also, the advent of easy and fast conferencing software can mitigate the disadvantages of remote work.

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 everyone 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, it doesn't need to contain all the minute details of the everyday life of the project, and it doesn't even need to be written down, 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 planning.

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, get a feel for the status of the entire project, and to create a sense of camaraderie within the team. This works very well for smaller teams (say, up to about 20 developers, many of which will have daily "stand-up" meetings to sync up at the beginning of each day. Different process models handle this "stand-up" meeting differently. For instance, plan-driven models don't require these meetings, depending on the team managers to communicate with each other. Agile processes often require all-hands daily meetings to facilitate constant team communication in a highly dynamic project environment.

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 what clients want to hear, what companies want to offer, and what managers and developers want to supply. "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 to tell your manager right away. Sure, they might be angry. But they'll be angrier when you end up a month behind and they didn't know it. Fred Brooks' 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, you need to examine and pick the right set of development tools. 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 than with a screwdriver. Even if you have to first obtain a hammer and then learn how to use it for the very first time, you are leveling up your toolkit and your skills in the process, investing in your ability to work more efficiently going forward. 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 are all experienced developers with facility on multiple platforms, tool choice is much easier. 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.

Conclusion

Software development is the heart of every software project, and it 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. This 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)[3]

References

Brooks, F. 1987. "No Silver Bullet." *IEEE Computer* 20 (4): 10–19. `www.inst.eecs.berkeley.edu/~maratb/readings/NoSilverBullet.html`.

Lammers, Susan. 1986. *Programmers At Work*. Redmond, WA: Microsoft Press.

[3] Lammers, Susan. 1986. *Programmers At Work*. Redmond, WA: Microsoft Press.

PART I

Models and Team Practices

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

The process of developing software is commonly described as the Software Development Lifecycle (SDLC). Every program, no matter how small, has a life cycle, broadly composed of the following 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 development process may combine multiple steps or iterate over a subset of steps repeatedly between releases, but, in one form or another, all development should encompass all of the above life cycle steps in order to create high-quality software. The two most common variations are *plan-based models*[1] and the newer *agile development* models.[2]

[1] Paulk, Mark C. 1995. *The Capability Maturity Model: Guidelines for Improving the Software Process. The SEI Series in Software Engineering*. Reading, Mass.: Addison-Wesley Pub. Co.

[2] Martin, Robert C. 2003. *Agile Software Development, Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall.