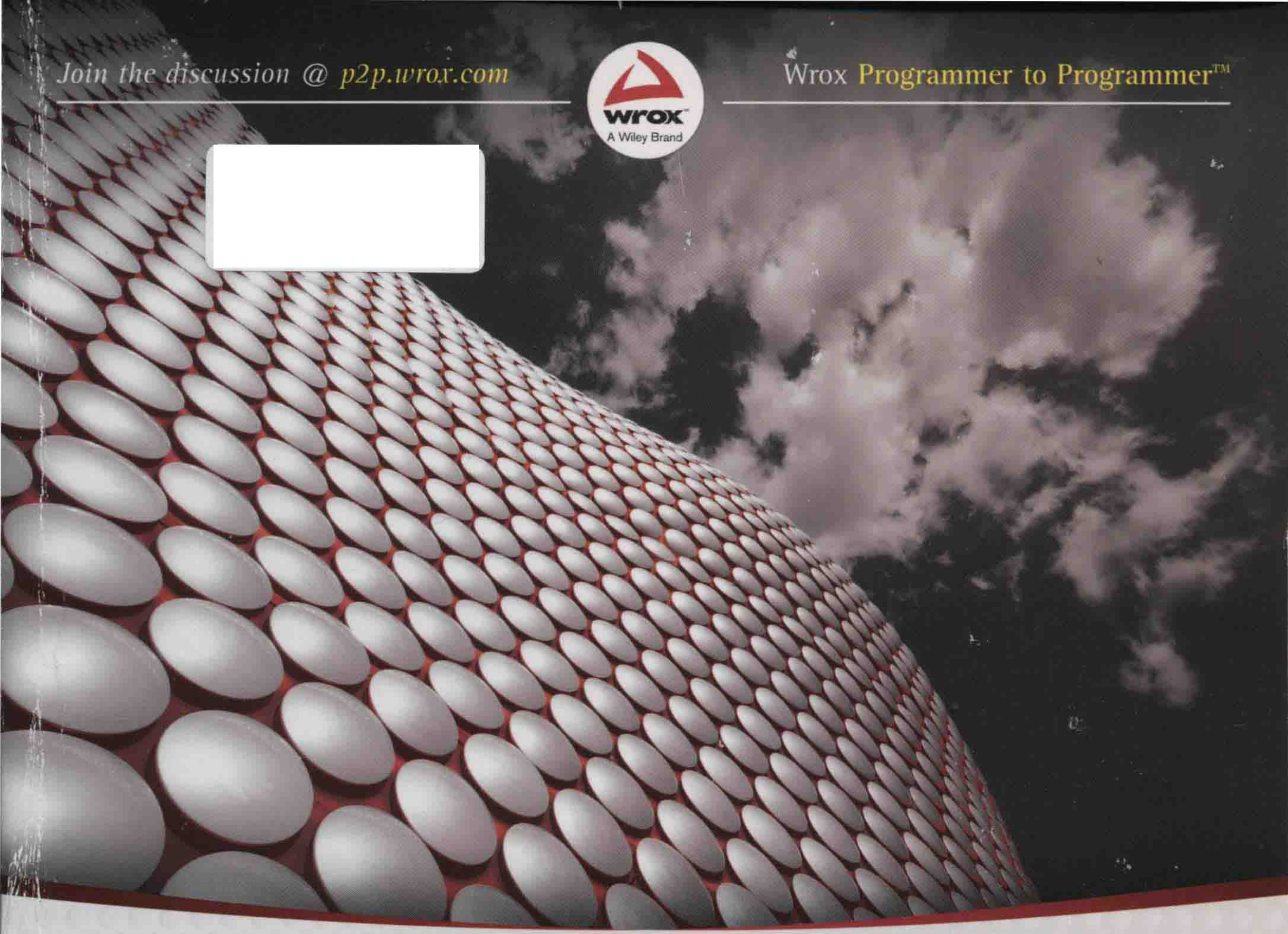
Join the discussion @ p2p.wrox.com

wrox A Wiley Brand

Wrox Programmer to Programmer™



Beginning Software Engineering

Rod Stephens

BEGINNING

Software Engineering

Rod Stephens

wrox™
A Wiley Brand

Beginning Software Engineering

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
`www.wiley.com`

Copyright © 2015 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-96914-4

ISBN: 978-1-118-96916-8 (ebk)

ISBN: 978-1-118-96917-5 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2015930533

BEGINNING SOFTWARE ENGINEERING

ABOUT THE AUTHOR

ROD STEPHENS started out as a mathematician, but while studying at MIT, he discovered how much fun programming is and he's been programming professionally ever since. During his career, he has worked on an eclectic assortment of applications in such fields as telephone switching, billing, repair dispatching, tax processing, wastewater treatment, concert ticket sales, cartography, and training for professional football players.

Rod has been a Microsoft Visual Basic Most Valuable Professional (MVP) for more than a decade and has taught introductory programming courses. He has written more than two dozen books that have been translated into languages from all over the world, and he's written more than 250 magazine articles covering Visual Basic, C#, Visual Basic for Applications, Delphi, and Java.

Rod's popular VB Helper website (`www.vb-helper.com`) receives several million hits per month and contains thousands of pages of tips, tricks, and example programs for Visual Basic programmers. His C# Helper website (`www.csharphelper.com`) contains similar material for C# programmers.

You can contact Rod at `RodStephens@CSharpHelper.com` or `RodStephens@vb-helper.com`.

ABOUT THE TECHNICAL EDITOR

BRIAN HOCHGURTEL has been doing .NET development for over ten years, and actually started his .NET experience with Rod Stephens when they wrote the Wiley book, *Visual Basic.NET and XML*, in 2002. Currently Brian works with C#, SQL Server, and SharePoint in Fort Collins, CO.

CREDITS

EXECUTIVE EDITOR
Robert Elliott

PROJECT EDITOR
Adaobi Obi Tuiton

TECHNICAL EDITOR
Brian Hochgurtel

PRODUCTION MANAGER
Kathleen Wisor

COPY EDITOR
San Dee Phillips

MANAGER OF CONTENT DEVELOPMENT & ASSEMBLY
Mary Beth Wakefield

MARKETING DIRECTOR
David Mayhew

MARKETING MANAGER
Carrie Sherrill

PROFESSIONAL TECHNOLOGY & STRATEGY DIRECTOR
Barry Pruett

BUSINESS MANAGER
Amy Knies

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Brent Savage

PROOFREADER
Sarah Kaikini, Word One

INDEXER
Johnna VanHoose Dinse

COVER DESIGNER
Wiley

COVER IMAGE
©iStock.com/Chris Hepburn

ACKNOWLEDGMENTS

THANKS TO BOB ELLIOTT, Adaobi Obi Tulton, San Dee Phillips, Katie Wisor, and all the others who worked so hard to make this book possible. (Adaobi was this book's project manager. You'll learn what a project manager does in Chapter 3. It's a bit different for writing a book but not as different as you might think. As always, many thanks for your hard work, Adaobi!)

Thanks also to technical editor Brian Hochgurtel for giving me the benefit of his valuable experience.

Special thanks to Mary Brodie at `gearmark.blogs.com` for letting me use her quote in Chapter 13, "Iterative Models."

INTRODUCTION

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the universe trying to build bigger and better idiots. So far the universe is winning.

—Rick Cook

With modern development tools, it's easy to sit down at the keyboard and bang out a working program with no previous design or planning, and that's fine under some circumstances. My VB Helper (`www.vb-helper.com`) and C# Helper (`www.csharphelper.com`) websites contain thousands of example programs written in Visual Basic and C#, respectively, and built using exactly that approach. I had an idea (or someone asked me a question) and I pounded out a quick example.

Those types of programs are fine if you're the only one using them and then for only a short while. They're also okay if, as on my websites, they're intended only to demonstrate a programming technique and they never leave the confines of the programming laboratory.

If this kind of slap-dash program escapes into the wild, however, the result can be disastrous. At best, nonprogrammers who use these programs quickly become confused. At worst, they can wreak havoc on their computers and even on those of their friends and coworkers.

Even experienced developers sometimes run afoul of these half-baked programs. I know someone (I won't give names, but I also won't say it wasn't me) who wrote a simple recursive script to delete the files in a directory hierarchy. Unfortunately, the script recursively climbed its way to the top of the directory tree and then started cheerfully deleting every file in the system. The script ran for only about five seconds before it was stopped, but it had already trashed enough files that the operating system had to be reinstalled from scratch. (Actually, some developers believe reinstalling the operating system every year or so is character-building. If you agree, perhaps this approach isn't so bad.)

I know another experienced developer who, while experimenting with Windows system settings, managed to set every system color to black. The result was a black cursor over a black desktop, displaying black windows with black borders, menus, and text. This person (who wasn't me this time) eventually managed to fix things by rebooting and using another computer that wasn't color-impaired to walk through the process of fixing the settings using only keyboard accelerators. It was a triumph of cleverness, but I suspect she would have rather skipped the whole episode and had her two wasted days back.

For programs that are more than a few dozen lines long, or that will be given to unsuspecting end users, this kind of free-spirited development approach simply won't do. To produce applications that are effective, safe, and reliable, you can't just sit down and start typing. You need a plan. You need ... <drumroll> ... software engineering.

This book describes software engineering. It explains what software engineering is and how it helps produce applications that are effective, flexible, and robust enough for use in real-world situations.

This book won't make you an expert systems analyst, software architect, project manager, or programmer, but it explains what those people do and why they are necessary for producing high-quality software. It also gives you the tools you need to start. You won't rush out and lead a 1,000-person effort to build a new air traffic control system for the FAA, but it can help you work effectively in small-scale and large-scale development projects. (It can also help you understand what a prospective future boss means when he says, "Yeah, we mostly use Scrum with a few extra XP techniques thrown in.")

WHAT IS SOFTWARE ENGINEERING?

A formal definition of software engineering might sound something like, "An organized, analytical approach to the design, development, use, and maintenance of software."

More intuitively, software engineering is everything you need to do to produce successful software. It includes the steps that take a raw, possibly nebulous idea and turn it into a powerful and intuitive application that can be enhanced to meet changing customer needs for years to come.

You might be tempted to restrict software engineering to mean only the beginning of the process, when you perform the application's design. After all, an aerospace engineer designs planes but doesn't build them or tack on a second passenger cabin if the first one becomes full. (Although I guess a space shuttle riding piggyback on a 747 sort of achieved that goal.)

One of the big differences between software engineering and aerospace engineering (or most other kinds of engineering) is that software isn't physical. It exists only in the virtual world of the computer. That means it's easy to make changes to any part of a program even after it is completely written. In contrast, if you wait until a bridge is finished and then tell your structural engineer that you've decided to add two extra lanes, there's a good chance he'll cackle wildly and offer you all sorts of creative but impractical suggestions for exactly what you can do with your two extra lanes.

The flexibility granted to software by its virtual nature is both a blessing and a curse. It's a blessing because it lets you refine the program during development to better meet user needs, add new features to take advantage of opportunities discovered during implementation, and make modifications to meet evolving business needs. It even allows some applications to let users write scripts to perform new tasks never envisioned by developers. That type of flexibility isn't possible in other types of engineering.

Unfortunately, the flexibility that allows you to make changes throughout a software project's life cycle also lets you mess things up at any point during development. Adding a new feature can break existing code or turn a simple, elegant design into a confusing mess. Constantly adding, removing, and modifying features during development can make it impossible for different parts of the system to work together. In some cases, it can even make it impossible to tell when the project is finished.

Because software is so malleable, design decisions can be made at any point up to the end of the project. Actually, successful applications often continue to evolve long after the initial release. Microsoft Word, for example, has been evolving for roughly 30 years. (Sometimes for the better, sometimes for the worse. Remember Clippy? I'll let you decide whether that change was for the better or for the worse, but I haven't seen him in a while.)

The fact that changes can come at any time means you need to consider the whole development process as a single, long, complex task. You can't simply "engineer" a great design, turn the programmers loose on it, and walk off into the sunset wrapped in the warm glow of a job well done. The biggest design decisions may come early, and software development certainly has stages, but those stages are linked, so you need to consider them all together.

WHY IS SOFTWARE ENGINEERING IMPORTANT?

Producing a software application is relatively simple in concept: Take an idea and turn it into a useful program. Unfortunately for projects of any real scope, there are countless ways that a simple concept can go wrong. Programmers may not understand what users want or need (which may be two separate things), so they build the wrong application. The program might be so full of bugs that it's frustrating to use, impossible to fix, and can't be enhanced over time. The program could be completely effective but so confusing that you need a PhD in puzzle-solving to use it. An absolutely perfect application could even be killed by internal business politics or market forces.

Software engineering includes techniques for avoiding the many pitfalls that otherwise might send your project down the road to failure. It ensures the final application is effective, usable, and maintainable. It helps you meet milestones on schedule and produce a finished project on time and within budget. Perhaps most important, software engineering gives you the flexibility to make changes to meet unexpected demands without completely obliterating your schedule and budget constraints.

In short, software engineering lets you control what otherwise might seem like a random whirlwind of chaos.

WHO SHOULD READ THIS BOOK?

Everyone involved in any software development effort should have a basic understanding of software engineering. Whether you're an executive customer specifying the software's purpose and features, an end user who will eventually spend time working with (and reporting bugs in) the finished application, a lead developer who keeps other programmers on track (and not playing too much Flow Free), or the guy who fetches donuts for the weekly meeting, you need to understand how all the pieces of the process fit together. A failure by any of these people (particularly the donut wallah) affects everyone else, so it's essential that everyone knows the warning signs that indicate the project may be veering toward disaster.

This book is mainly intended for people with limited experience in software engineering. It doesn't expect you to have any previous experience with software development, project management, or programming. (I suspect most readers will have some experience with donuts, but that's not necessary, either.)

Even if you have some familiarity with those topics, particularly programming, you may still find this book informative. If you've been focusing only on the pieces of a project assigned to you, you still need to learn about how the pieces interact to help guide the project toward success.

For example, I had been working as a programmer for several years and even taken part in some fairly large development efforts before I took a good look at the development process as a whole. I knew other people were writing use cases and deployment plans, but my focus was on my piece of the project. It wasn't until later, when I started taking a higher-level role in projects that I actually started to see the entire process.

This book does not explain how to program. It does explain some techniques programmers can use to produce code that is flexible enough to handle the inevitable change requests, easy to debug (at least your code will be), and easy to enhance and maintain in the future (more change requests), but they are described in general terms and don't require you to know how to program.

If you don't work in a programming role, for example if you're an end user or a project manager, you'll hopefully find that material interesting even if you don't use it directly. You may also find some techniques surprisingly applicable to nonprogramming problems. For example, techniques for generating problem-solving approaches apply to all sorts of problems, not just programming decisions. (You can also ask developers, "Are you using assertions and gray-box testing methods before unit testing?" just to see if they understand what you're talking about. Basically, you're using gray-box testing to see if the developers know what gray-box testing is. You'll learn more about that in Chapter 8, "Testing.")

APPROACH

This book is divided into two parts. The first part describes the basic tasks you need to complete and deliver useful software. Things such as design, programming, and testing. The book's second part describes some common software development models that use different techniques to perform those tasks.

Before you can begin to work on a software development project, however, you need to do some preparation. You need to set up tools and techniques that help you track your progress throughout the project. Chapter 1, "Software Engineering from 20,000 Feet," describes these "before-the-beginning" activities.

After you have the preliminaries in place, there are many approaches you can take to produce software. All those approaches have the same goal (making useful software), so they must handle roughly the same tasks. These are things such as gathering requirements, building a plan, and actually writing the code. The first part of this book describes these tasks. Chapter 1 explains those tasks at a high level. Chapters 2 through 11 provide additional details about what these tasks are and how you can accomplish them effectively.

The second part of the book describes some of the more popular software development approaches. All these models address the same issues described in the earlier chapters but in different ways. Some focus on predictability so that you know exactly what features will be provided and when. Others focus on creating the most features as quickly as possible, even if that means straying from the original design. Chapters 12 through 14 describe some of the most popular of these development models.

That's the basic path this book gives you for learning software engineering. First learn the tasks you need to complete to deliver useful software. Then learn how different models handle those tasks.

However, many people have trouble learning by slogging through a tedious enumeration of facts. (I certainly do!) To make the information a bit easier to absorb, this book includes a few other elements.

Each chapter ends with exercises that you can use to see if you were paying attention while you read the chapter. I don't like exercises that merely ask you to repeat what is in the chapter. (Quick, what are some advantages and disadvantages of the ethereal nature of software?) Most of the exercises ask you to expand on the chapter's main ideas. Hopefully, they'll make you think about new ways to use what's explained in the chapter.

Sometimes, the exercises are the only way I could sneak some more information into the chapter that didn't quite fit in any of its sections. In those cases, the questions and answers provided in Appendix A are like extended digressions and thought experiments than quiz questions.

I strongly recommend that you at least skim the exercises and think about them. Then ask yourself if you understand the solutions. All the solutions are included in Appendix A, "Solutions to Exercises."

WHAT THIS BOOK COVERS (AND WHAT IT DOESN'T)

This book describes software engineering, the tasks that you must perform to successfully complete a software project, and some of the most popular developer models you can use to try to achieve your goals. It doesn't cover every last detail, but it does explain the overall process so that you can figure out how you fit into the process.

This book does not explain every possible development model. Actually, it barely scratches the surface of the dozens (possibly hundreds) of models that are in use in the software industry. This book describes only some of the most popular development approaches and then only relatively briefly.

If you decide you want to learn more about a particular approach, you can turn to the hundreds of books and thousands of web pages written about specific models. Many development models also have their own organizations with websites dedicated to their promotion. For example, see `www.extremeprogramming.org`, `agilemanifesto.org`, and `www.scrum.org`.

This book also isn't an exhaustive encyclopedia of software development tricks and tips. It describes some general ideas and concepts that make it easier to build robust software, but its focus is on higher-level software engineering issues, so it doesn't have room to cover all the clever techniques developers use to make programs better. This book also doesn't focus on a specific programming language, so it can't take advantage of language-specific tools or techniques.

WHAT TOOLS DO YOU NEED?

You don't need any tools to read this book. All you need is the ability to read the book. (And perhaps reading glasses. Or perhaps a text-to-speech tool if you have an electronic version that you want to "read." Or perhaps a friend to read it to you. Okay, I guess you have several options.)

To actually participate in a development effort, you may need a lot of tools. If you're working on a small, one-person project, you might need only a programming environment such as Visual Studio, Eclipse, RAD Studio, or whatever. For larger team efforts you'll also need tools for project management, documentation (word processors), change tracking, software revision tracking, and more. And, of course, you'll need other developers to help you. This book describes these tools, but you certainly don't need them to read the book.

CONVENTIONS

To help you get the most from the text and keep track of what's happening, I've used several conventions throughout the book.

SPLENDID SIDEBARS

Sidebars such as this one contain additional information and side topics.

WARNING *Boxes like this one hold important information that is directly relevant to the surrounding text. There are a lot of ways a software project can fail, so these warn you about "worst practices" that you should avoid.*

NOTE *These boxes indicate notes, tips, hints, tricks, and asides to the current discussion. They look like this.*

As for styles in the text:

- Important words are *highlighted* when they are introduced.
- Keyboard strokes are shown like this: Ctrl+A. This one means you should hold down the Ctrl key (or Control or CTL or whatever it's labeled on your keyboard) and press the A key.
- This book includes little actual program code because I don't know what programming languages you use (if any). When there is code, it is formatted like the following.

```
// Return true if a and b are relatively prime.
private bool AreRelativelyPrime(int a, int b)
{
    // Only 1 and -1 are relatively prime to 0.
    if (a == 0) return ((b == 1) || (b == -1));
    if (b == 0) return ((a == 1) || (a == -1));

    int gcd = GCD(a, b);
    return ((gcd == 1) || (gcd == -1));
}
```

(Don't worry if you can't understand the code. The text explains what it does.)