SECOND EDITION

BEGINNING Software Engineering

Rod Stephens

WILEY

BEGINNING SOFTWARE ENGINEERING

Continues

BEGINNING Software Engineering

BEGINNING

Software Engineering

Second Edition

Rod Stephens

WILEY

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada and the United Kingdom.

ISBN: 978-1-119-90170-9
ISBN: 978-1-119-90172-3 (ebk.)
ISBN: 978-1-119-90171-6 (ebk.)

Library of Congress Control Number: 2022944804

Cover Image and Design: Wiley

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. He's a long-time developer, instructor, and author who has written more than 250 magazine articles and 35 books that have been translated into many different languages.

During his career, Rod 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, optometry, and training for professional football teams. (That's US football, not one of the kinds with the round ball. Or the kind with three downs. Or the kind with an oval field. Or the indoor kind. Let's just say NFL and leave it at that.)

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

You can contact Rod at `RodStephens@csharphelper.com`.

ABOUT THE TECHNICAL EDITOR

John Mueller is a freelance author and technical editor. He has writing in his blood, having produced 122 books and more than 600 articles to date. The topics range from networking to artificial intelligence and from database management to heads-down programming. Some of his current books include discussions of data science, machine learning, and algorithms. He also writes about computer languages such as C++, C#, and Python. His technical editing skills have helped more than 70 authors refine the content of their manuscripts. John has provided technical editing services to a variety of magazines, performed various kinds of consulting, and he writes certification exams. Be sure to read John's blog at `http://blog.johnmuellerbooks.com`. You can reach John on the Internet at `John@JohnMuellerBooks.com`. John also has a website at `www.johnmuellerbooks.com`.

ACKNOWLEDGMENTS

Thanks to David Clark, Christine O'Connor, Kenyon Brown, Judy Flynn, Barath Kumar Rajasekaran, and all of the others who worked so hard to make this book possible. David was this book's project manager. You'll learn what a project manager does in Chapter 4. It's a bit different for writing a book but not as different as you might think. Many thanks for your hard work, David!

Thanks also to technical editor and longtime friend John Mueller for giving me the benefit of his valuable experience. You can see what John's up to at `www.johnmuellerbooks.com`.

Special thanks to Mary Brodie at `https://gearmark.blogs.com` for letting me use her quote at the beginning of Chapter 18, "Iterative Models."

CONTENTS

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, then 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 that 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 employer 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 that 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 and lift it three feet higher above the water, 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 requirements. Some applications even allow users to write scripts to perform new tasks never envisioned by the 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 40 years, sometimes for the better, sometimes for the worse. (If you don't remember Clippy, search online to learn the tragic tale.)

The fact that changes can come at any time means that 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 ride 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 that 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 importantly, 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 *Minecraft*), or the guy who fetches donuts for the weekly meeting, you need to understand how all of 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. Most software developers focus primarily on one piece of the puzzle and don't really understand the rest of the process. It's worth learning how the pieces interact to help guide the project toward success.

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, is easy to debug (at least your code will be), and is 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 that are 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 13, "Testing.")

APPROACH

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

The third and final part of the book contains two bonus chapters, "Software Ethics" and "Future Trends," that provide useful information for any software developer but that didn't fit in well with the earlier parts of the book. After those come the Appendix, which contains the answers to the chapters' exercises, and the Glossary.

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. If you don't keep track of your progress, it's shockingly easy to fall hopelessly far behind. Chapter 1, "Software Engineering from 20,000 Feet," provides a high-level overview. Chapter 2, "Before the Beginning," and Chapter 3, "The Team," describe some of the other setup tasks that you need to start before the more concrete development can really get rolling.

After you have the preliminaries in place, there are many approaches that you can take to produce software. All of 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 4 through 16 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 of 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 17 through 19 describe some of the most popular of these development models.

Chapter 20 discusses software ethics. Software presents some unique ethical dilemmas and artificial intelligence (AI) provides a framework for some situations that are interesting if somewhat unlikely. Finally, in Chapter 21, I make some (probably foolish) predictions about software engineering trends.

That's the basic path this book gives you for learning software engineering. First learn the tasks that you need to complete to deliver useful software. Next, learn how different models handle those tasks. Then finish with some more thoughtful material.

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 chapters.

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 the answers provided in the Appendix are more 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 of the solutions are included in the Appendix.

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 that 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 that 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`, `https://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 of the clever techniques that 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" while driving. 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 Jupyter Notebook, 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.

NOTE *For anyone who's adopted the book as part of teaching a software engineering course, I've provided instructor supplemental materials (ISM) that you can use. Go to the book details page. Click the "Related Resources" link (or scroll down the page) to navigate to the "View Instructor Companion Site" link. Click the link to open the Instructor Companion Site page. To access and download the ISM, you'll need to sign in to request them. If you don't have a Wiley account, you'll need to sign up to create one.*

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 a particular piece of code. The text explains what it does.)
- Filenames, URLs, and the occasional piece of code within the text are shown like this: `www.csharphelper.com`.

ERRATA

I've done my best to avoid errors in this book, and this book has passed through the word processors of a small army of editors and technical reviewers. However, no nontrivial project is ever completely without mistakes. (That's one of the more important lessons in this book.) The best I can hope for is that any remaining errors are small enough that they don't distract you from the meaning of the text.

If you find an error in one of my books (like a spelling mistake, broken piece of code, or something that just doesn't make sense), I would be grateful for your feedback. Sending in errata may save other readers hours of frustration. At the same time, you'll be helping me provide even higher quality information.

To find the errata page for this book, go to `www.wiley.com` and search for the book. Then, on the book details page, click the Errata link. On this page you can view all of the errata submitted for this book.

If you don't spot "your" error on the Book Errata page, please email it to our Customer Service Team at `wileysupport@wiley.com` with the subject line "Possible Book Errata Submission."

IMPORTANT URLs

Here's a summary of important URLs related to this book:

- `RodStephens@CSharpHelper.com`—My email address. I hope to hear from you!
- `www.CSharpHelper.com`—My C# website, which contains thousands of tips, tricks, and examples for C# developers.
- `www.vb-helper.com`—My Visual Basic website, which contains thousands of tips, tricks, and examples for Visual Basic developers.

CONTACTING THE AUTHOR

If you have questions, suggestions, comments, just want to say hi, want to exchange cookie recipes, or whatever, email me at `RodStephens@CSharpHelper.com`. I can't promise that I'll be able to help you with every problem, but I do promise to try. (And I have some pretty good cookie recipes.)

DISCLAIMER

Software engineering isn't always the most exciting topic, so in an attempt to keep you awake, I picked some of the examples in this book for interest or humorous effect. (If you keep this book on your nightstand as a last-ditch insomnia remedy, then I've failed.)

I mean no disrespect to any of the many talented software engineers out there who work long weeks (despite the call for sustainable work levels) to produce top-quality applications for their customers. (As for the untalented software engineers out there, their work can speak for them better than I can.)

I also don't mean to discount any of the development models described in this book or the people who worked on or with them. Every one of them represents a huge amount of work and research, and all of them have their places in software engineering, past or present.

Because this book has limited space, I had to leave out many software development methodologies and programming best practices. Even the methodologies that *are* described are not covered in full detail because there just isn't room.

Finally, I mean no disrespect to people named Fred, or anyone else for that matter. (Except for one particular Fred, who I'm sure retired from software development long ago.)

So get out your reading glasses, grab your favorite caffeinated beverage, and prepare to enter the world of software engineering. Game on!

PART I
Software Engineering Step-by-Step

Software and cathedrals are much the same. First we build them, then we pray.

—*Samuel Redwine*

In principle, software engineering is a simple two-step process: (1) Write a best-selling program, and then (2) buy expensive toys with the profits. Unfortunately, the first step can be rather difficult. Saying "write a best-selling program" is a bit like telling an author, "Write a best-selling book," or telling a baseball player "triple to left." It's a great idea, but knowing the goal doesn't actually help you achieve it.

To produce great software, you need to handle a huge number of complicated tasks, any one of which can fail and sink the entire project. Over the years people have developed a multitude of methodologies and techniques to help keep software projects on track. Some of these, such as the *waterfall* and *V-model* approaches, use detailed requirement specifications to exactly define the desired results before development begins. Others, such as *Scrum* and *agile techniques*, rely on fast-paced incremental development with frequent feedback to keep a project on track. Still other techniques, such as cowboy coding and extreme programming, sound more like action-adventure films than software development techniques. (I'll say more about these in Part II, "Process Models.")

Different development methodologies use different approaches, but they all perform roughly the same tasks. They all determine what the software should do and how it should do it. They generate the software, remove bugs from the code (some of the bugs, at least), make sure the software does more or less what it should, and deploy the finished result.

> ***NOTE*** *I call these basic items "tasks" and not "stages" or "steps" because different software engineering approaches tackle them in different ways and at different times. Calling them "stages" or "steps" would probably be misleading because it would imply that all projects move through the stages in the same predictable order and that's not true.*

The chapters in the first part of this book describe those basic tasks that any successful software project must handle in some way. They explain the main steps in software development and describe some of the myriad ways a project can fail to handle those tasks. (The second part of the book explains how different approaches such as waterfall and agile handle those tasks.)

The first chapter in this part of the book provides an overview of software development from a high level. The subsequent chapters explain the pieces of the development process in greater detail.

1 Software Engineering from 20,000 Feet

If you fail to plan, you are planning to fail.

—*Benjamin Franklin*

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. The other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

—*C.A.R. Hoare*

What You Will Learn in This Chapter:

- The basic steps required for successful software engineering
- Ways in which software engineering differs from other kinds of engineering
- How fixing one bug can lead to others
- Why it is important to detect mistakes as early as possible

In many ways, software engineering is a lot like other kinds of engineering. Whether you're building a bridge, an airplane, a nuclear power plant, or a new and improved version of Sudoku, you need to accomplish certain tasks. For example, you need to make a plan, follow that plan, heroically overcome unexpected obstacles, and hire a great band to play at the ribbon-cutting ceremony.

The following sections describe the steps you need to take to keep a software engineering project on track. These are more or less the same for any large project, although there are some important differences that are specific to software engineering. Later chapters in this book provide a lot more detail about these tasks.

REQUIREMENTS GATHERING

No big project can succeed without a plan. Sometimes a project doesn't follow the plan closely, but every big project must have a plan. The plan tells project members what they should be doing, when and how long they should be doing it, and most important, what the project's goals are. They give the project direction.

One of the first steps in a software project is figuring out the requirements. You need to find out what the customers want and what the customers need. Depending on how well-defined the user's needs are, this chore can be time-consuming.

WHO'S THE CUSTOMER?

Sometimes, it's easy to tell who the customer is. If you're writing software for another part of your own company, it may be obvious who the customers are. In that case, you can sit down with them and talk about what the software should do.

In other cases, you may have only a vague notion of who will use the finished software. For example, if you're creating a new online card game, it may be hard to identify the customers until after you start marketing the game.

Sometimes, *you* may even be the customer. I write software for myself all the time. This has a lot of advantages. For example, I know exactly what I want (usually) and I know more or less how hard it will be to provide different features. (Unfortunately, I also sometimes have a hard time saying no to myself, so projects can drag on for a lot longer than they should.)

In any project, you should try to identify your customers and interact with them as much as possible so that you can design the most useful application possible.

After you determine the customers' wants and needs (which are not always the same), you can turn them into requirements documents. Those documents tell the customers what they will be getting, and they tell the project members what they will be building.

NOTE *I refer to requirements a lot in this book. In general, when I say requirements, I mean the officially sanctioned requirements as recorded in the requirements documents and later carved in marble on the project's memorial. Anything else (such as customer suggestions, developer complaints, and management wishful thinking) are not part of the requirements until they are approved by the powers that be.*

Throughout the project, both customers and team members can refer to the requirements to see if the project is heading in the right direction. If someone suggests that the project should include a video tutorial, you can see if that was included in the requirements. If this is a new feature, you might allow that change if it would be useful and wouldn't mess up the rest of the schedule. If that request doesn't make sense, either because it wouldn't add value to the project or you can't do it with the time you have, then you may need to defer it for a later release.

CHANGE HAPPENS

Although there are some similarities between software and other kinds of engineering, the fact that software doesn't exist in any physical way means there are some major differences as well. Because software is so malleable, users frequently ask for new features up to the day before the release party. They ask developers to shorten schedules and request last-minute changes such as switching database platforms or even hardware platforms. (Yes, both of those have happened to me.) "The program is just 0s and 1s," they reason. "The 0s and 1s don't care whether they run on an Android tablet or an iPhone, do they?"

In contrast, a company wouldn't ask an architectural firm to move a new convention center across the street at the last minute; a city transportation authority wouldn't ask the builder to add an extra lane to a freeway bridge right after it opens; and no one would try to insert an atrium level at the bottom of a newly completed 90-story building.

HIGH-LEVEL DESIGN

After you know the project's requirements, you can start working on the high-level design. The high-level design includes such things as decisions about what platform to use (such as desktop, laptop, tablet, or phone), what data design to use (such as direct access, 2-tier, or 3-tier), and what interfaces with other systems to use (such as external purchasing systems or a payroll system hosted in the cloud).

The high-level design should also include information about the project architecture at a relatively high level. You should break the project into the large chunks that handle the project's major areas of functionality. Depending on your approach, this may include a list of the modules that you need to build or a list of families of classes.

For example, suppose you're building a system to manage the results of ostrich races. You might decide the project needs the following major pieces:

- Database (to hold the data)
- Classes (for example, Race, Ostrich, Jockey, and Wager classes)

- User interfaces (to enter Ostrich and Jockey data, enter race results, calculate odds, produce result reports, and create new races)
- External interfaces (to send information and spam to participants and fans via email, text message, voicemail, pager, carrier pigeon, and anything else we can think of)

You should make sure the high-level design covers every aspect of the requirements. It should specify what the pieces do and how they should interact, but it should include as few details as possible about *how* the pieces do their jobs.

TO DESIGN OR NOT TO DESIGN, THAT IS THE QUESTION

At this point, fans of extreme programming, Scrum, and other incremental development approaches may be rolling their eyes, snorting in derision, and muttering about how they don't need high-level designs.

Let's defer this argument until Chapter 6, "High-Level Design," which talks about high-level design in greater detail. For now, I'll just claim that every design methodology needs design, even if it doesn't come in the form of a giant written design specification carved into a block of marble.

LOW-LEVEL DESIGN

After your high-level design breaks the project into pieces, you can assign those pieces to groups within the project so they can work on low-level designs. The low-level design includes information about *how* that piece of the project should work. The design doesn't need to give every last nitpicky detail necessary to implement the project's major pieces, but it should give enough guidance to the developers who will implement those pieces.

For example, the ostrich racing application's database piece would include an initial design for the database. It should sketch out the tables that will hold the race, ostrich, and jockey information using proper first, second, and third normal forms. (You can argue about whether it needs higher levels of normalization.)

At this point you will also discover interactions between the different pieces of the project that may require changes here and there. For example, while working on the ostrich project's external interfaces, you may decide to add a new table to hold email, text messaging, and other information for fans. Or you may find that the printing module will be easier if you add a new stored procedure to the database design.

DEVELOPMENT

After you've created the high- and low-level designs, it's time for the programmers to get to work. (Actually, the programmers should have been hard at work gathering requirements, creating the high-level designs, and refining them into low-level designs, but development is the part that many programmers enjoy the most, so that's often where they think the "real" work begins.) The programmers continue refining the low-level designs until they know how to implement those designs in code.

(In fact, in one of my favorite development techniques, you basically just keep refining the design to give more and more detail until it would be easier to just write the code instead. Then you do exactly that.)

As the programmers write the code, they test it to make sure it doesn't contain any bugs.

At this point, any experienced developers should be snickering if not actually laughing out loud. It's a programming axiom that no nontrivial program is completely bug-free. So let me rephrase the previous paragraph.

As the programmers write the code, they test it to find and remove as many bugs as they reasonably can.

TESTING

Effectively testing your own code is extremely hard. If you just wrote the code, you obviously didn't insert bugs intentionally. If you knew there was a bug in the code, you would have fixed it before you wrote it. That idea often leads programmers to assume their code is correct (I guess they're just naturally optimistic), so they don't always test it as thoroughly as they should.

Even if a particular piece of code is thoroughly tested and contains no (or few) bugs, there's no guarantee that it will work properly with the other parts of the system.

One way to address both of these problems (developers don't test their own code well and the pieces may not work together) is to perform different kinds of tests. First developers test their own code. Then testers who didn't write the code test it. After a piece of code seems to work properly, it is integrated into the rest of the project, and the whole thing is tested to see if the new code broke anything.

Any time a test fails, the programmers dive back into the code to figure out what's going wrong and how to fix it. After any repairs, the code goes back into the queue for retesting.

A SWARM OF BUGS

At this point you may wonder why you need to retest the code. After all, you just fixed it, right?

Unfortunately, fixing a bug often creates a new bug. Sometimes the bug fix is incorrect. Other times it breaks another piece of code that depended on the original buggy behavior. In that case, the known bug hides an unknown bug.

Still other times the programmer might change some correct behavior to a different correct behavior without realizing that some other code depended on the original correct behavior. (Imagine if someone switched the arrangement of your hot- and cold-water faucets. Either arrangement would work just fine, but you may get a nasty surprise the next time you take a shower.)

Anytime you change the code, whether by adding new code or fixing old code, you need to test it to make sure everything works as it should.

Unfortunately, you can never be certain that you've caught every bug. If you run your tests and don't find anything wrong, that doesn't mean there are no bugs; it just means you haven't found them. As programming pioneer Edsger W. Dijkstra said, "Testing shows the presence, not the absence of bugs." (This issue can become philosophical. If a bug is never detected, is it still a bug?)

The best you can do is test and fix bugs until they occur at an acceptably low rate. If bugs don't bother users too frequently or too severely when they do occur, then you're ready to move on to deployment.

COUNTING BUGS

Suppose requirements gathering, high-level design, low-level design, and development works like this: Every time you make a decision, the next task in the sequence includes two more decisions that depend on the first one. For example, when you make a requirements decision, the high-level design includes two decisions that depend on it. (This isn't exactly the way it works, but it's not as ridiculous as you might wish.)

Now suppose you made a mistake during requirements gathering. (The customer said the application had to support 30 users with a 5-second response time, but you heard 5 users with a 30-second response time.)

If you detect the error during the requirements gathering phase, you need to fix only that one error. But how many incorrect decisions could depend on that one mistake if you don't discover the problem until after development is complete?

The one mistake in requirements gathering leads to two decisions in high-level design that could be incorrect.

Each of the two possible mistakes in high-level design leads to two new decisions in low-level design that could also be wrong, giving a total of 2 × 2 = 4 possible mistakes in low-level design.

Each of the four suspicious low-level design decisions lead to two more decisions during development, giving a total of 4 × 2 = 8 possible mistakes during development.

Adding up all the mistakes in requirements gathering, high-level design, low-level design, and development gives a total of 1 + 2 + 4 + 8 = 15 possible mistakes. Figure 1.1 shows how the potential mistakes propagate.

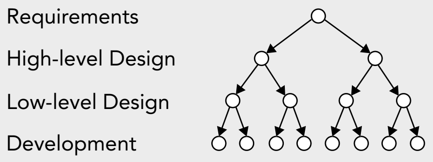

FIGURE 1.1: The circles represent possible mistakes at different stages of development. One early mistake can lead to many later mistakes.

To make matters worse, you usually won't know that all of these decisions were related. Just because you find some of the 15 bugs doesn't mean you'll know that the others exist.

In this example, you have 15 times as many decisions to track down, examine, and possibly fix than you would have if you had discovered the mistake right away during requirements gathering. That leads to one of the most important rules of software engineering:

The longer a bug remains undetected, the harder it is to fix.

Some people think of testing as something you do after the fact to verify that the code you wrote is correct. Actually, testing is critical at every stage of development to ensure the resulting application is usable.

DEPLOYMENT

Ideally, you roll out your software, the users are overjoyed, and everyone lives happily ever after. If you've built a new variant of Tetris and you release it on the Internet, your deployment may actually be that simple.

Often, however, things don't go so smoothly. Deployment can be difficult, time-consuming, and expensive. For example, suppose you've written a new billing system to track payments from your company's millions of customers. Deployment might involve any or all of the following:

- New computers for the backend database
- A new network
- New computers for the users
- User training
- On-site support while the users get to know the new system
- Parallel operations while some users get to know the new system and other users keep using the old system
- Special data maintenance chores to keep the old and new databases synchronized
- Massive bug fixing when the 250 users discover dozens or hundreds of bugs that testing didn't uncover
- Other nonsense that no one could possibly predict