CHAPTER 4

The Nine Practices

Building software is complex—perhaps the most complex activity that humans can engage in. Writing software is a discipline that requires a range of skills and practices to accomplish successfully.

It's easy to get it wrong because the virtual world is so different from the physical world. In the physical world, we can easily understand what it takes to build something but in the virtual world this can be much more difficult to see and understand. The software development profession is just starting to figure things out, much like where the medical profession was a few hundred years ago.

It was less than two hundred years ago that the medical community laughed at Ignaz Semmelweis for proposing that microscopic creatures could cause disease. How could something as trivial as washing your hands before surgery make the difference between life and death to your patient?

The medical community held this view in part because germ theory did not exist yet and partly because they were at that time actively trying to dispel the myth that invisible spirits caused disease (often truth and myth share a lot in common). Therefore, the practice of washing your hands before performing surgery wasn't considered essential.

Battlefield surgeons in the Civil War knew about germ theory but they argued they had no time to sterilize their instruments. If a soldier needed an amputation they didn't have time to wash the saw. But when the medical community looked at the success rates of battlefield surgery they discovered that in many cases more men died of infection and disease in the Civil War than died on the battlefield, so medicine had to rethink its position.

When we understand germ theory, we understand *why* we have to wash *all* the instruments. This is the difference between following a practice (the *act*

of sterilizing a specific instrument) and following a principle (the *reason for* sterilizing all instruments).

And the thing about the following software development practices, just like sterilization, is that we have to get it *all* right for *any* of it to work. If we happen to miss one of those things… one germ can kill your patient, and one bug can kill your application. And in this we require discipline.

I don't believe there is one right way to develop software just as there's no one right way to heal a patient, create art, or build a bridge. I don't believe in the "one true way" of anything, but especially not for programming.

Still, having worked with thousands of developers, I've seen first-hand how we constantly reinvent the wheel. Software development has attracted a range of people from all backgrounds, and that has brought many fresh perspectives to creating software. At the same time, the huge diversity among developers can be a real problem. Enterprise software development involves enormous attention to detail and a lot of coordination among those involved. We must have some shared understanding, a common set of practices, along with a shared vocabulary. We must arrive at a common set of goals and be clear that we value quality and maintainability.

Some developers are more effective than others, and I've spent most of my life trying to discover what makes these extraordinary developers so good. If we understand what they understand, learn some principles and practices, then we can achieve similar extraordinary results.

But where to begin?

Software design is a deep and complex subject and to do it justice requires a great deal of background theory that would need several books to explain. Furthermore, some key concepts are not yet well understood among all developers and many of us are still struggling to understand the context for software development.

In many ways, legacy code has come about because we've carried the notion that the quality of our code doesn't matter—all that matters is that software does what it's supposed to do.

But this is a false notion. If software is to be used it will need to be changed, so it must be written to be changeable. This is not how most software has been written. Most code is intertwined with itself so it's not independently deployable or extendable and that makes it expensive to maintain. It works for the purpose it was intended for, but because of the way it was written it's

hard to change, so people hack away at it, making it even harder and more costly to work with in the future.

We want to drop the cost of ownership of software. According to Barry Boehm[1] of the University of Southern California, it often costs 100 times more to find and fix a bug after delivery than it would cost during requirements and design. We have to find ways to drop the cost of supportability by making code easier to work with. If we want to drop the cost of ownership for software then we must pay attention to how we build it.

What Experts Know

Experts organize their knowledge in specific ways. They often have their own vocabulary to describe their key distinctions. They use metaphor and analogy, and have formed key beliefs around their experiences. Their *context for understanding* is different from the rest of us.

All of the techniques experts use are learnable skills. This means that when you understand what experts do and do what they do, you're likely to get the same results.

Expert software developers, the ones who are getting not just incrementally better but hugely better results, think about software development differently than the rest of us. They pay attention to technical practices and code quality. They understand what's important and what's not.

Most importantly, expert software developers hold themselves to higher standards than the rest of us.

I was surprised to find that the best developers I know are also the neatest. I figured fast coders had to be sloppy coders, but what I discovered was just the opposite. The fastest programmers I've met paid particular attention to keeping their code easy to work with. They don't just declare *instance variables* at the top of their classes, they list them in alphabetical order (or however else it makes sense), they constantly rename methods and move them around until their right home is found, and they immediately delete dead code that's not being used.

Even after noticing this correlation between these fast coders and how neat and tidy their code was, it took me a while to recognize the causal relationship between these two things. These people weren't faster *in spite* of keeping code

1. Boehm, Barry. Basili, Victor R. "Software Defect Reduction Top 10 List." Software Management, January 2001

quality high, they were faster *because* they kept their code quality high. Realizing this affects how we think about developing software.

Most people recognize that a well thought out approach to solving a problem can pay back over the long term. What most people don't realize is the payback is usually much faster than expected. In the physical world we recognize quality as a desirable attribute that we're willing to pay more for. Higher quality physical things tend to last longer and are therefore more expensive. But the virtual world is different.

In the virtual world, a focus on quality is always less costly to execute in the long term and often also in the short term. This doesn't mean developers shouldn't make compromises at times, but when they do they should also recognize the price they'll pay every time they'll have to go back and work with poor code. If that price is high then they may want to go back and clean up the code before making further enhancements.

Business takes a cost-benefit approach and software should be no different. Like any asset, software must be maintained so that it doesn't become a liability.

Shu-Ha-Ri

Mastery involves more than skill and ability. The Japanese martial art Aikido defines three stages of mastery: *Shu*, *Ha*, and *Ri*.

Shu is the form, the explicit knowledge. "Wax-on, wax-off" from the movie *The Karate Kid* is an example of the Shu stage of learning. Daniel, the young disciple in that movie, was told to wax cars with a circular movement. He wasn't told why or how it would prepare him to reach his goals. Once he mastered the form he was shown why.

Often, people learn Agile as a set of rules, as do's and don'ts. That is just the first stage of learning, yet many people feel that once they learn some rules they're ready to do Agile.

But complex activities, like developing software, are hard to pin down with rules. There's a lot of contraindications in software where the best approach in one situation could be a bad approach in another situation. As a result, there's typically a long learning curve to become a software developer.

The reason one starts with Shu is that the theory behind the practices isn't obvious. In martial arts, to defeat your opponent you must do more than understand theory, you must put theory into practice. In Aikido, this is called *Ha.* The same thing is true in software development. To be successful, we

must know the theories, the principles underlying the practices, in order to put the practices to good use.

You can't learn Ha prescriptively, as a set of rules. It has to come from experience but it is possible to learn some from other people's experiences.

Once you've used the practices, understand the underlying theory at a deep level, then practice and theory begin to dissolve and we approach the highest level of mastery in Aikido, *Ri*. This is the realm of true mastery that can only be obtained through ongoing study.

In his book *Outliers [Gla08]*, Malcolm Gladwell suggests that after 10,000 hours of practice we become natural in a domain that requires intellectual rigor. We no longer have to think about it because it's almost second nature. If truly mastering any complex activity takes about 10,000 hours then software development is no exception.

Pablo Picasso understood this. He learned the rules of painting so he could break them. He created paintings unlike anyone who came before him, but few people know that Picasso was trained as a classical painter. He could paint a painting in the style of the classical masters and spent most of his life acquiring those skills. But he wasn't satisfied. He went on to transcend those skills and arrive at something else. To break the rules and break new ground, we first have to master the rules.

The same is true with software development. There are a lot of rules and constraints when building software and there's also technique. Like any other human creation, a computer program is a model of something. We're used to *physical* models, but programs can also be *behavioral* models.

In order to accurately model something we must first understand what it is we're modeling and we must also understand what modeling skills or techniques are available to us. I find it useful to break out these techniques into two categories: principles and practices.

First Principles

First principles were originally described by Marcus Aurelius, when he discussed the Golden Rule, which states "Do unto others as you'd have them do unto you." The reason the Golden Rule is a first principle is that much of our law, our society, even our culture is based on that one simple statement. You can infer other principles from first principles.

The Golden Rule is an overarching first principle in the law. It's foundational to the pursuit of justice. Consider what the law would be like without the

Golden Rule, if it were every person for themselves. Understanding and agreeing to the right principles is central to the success of any discipline.

Software development doesn't yet have the equivalent of the Golden Rule or the Hippocratic Oath. We're still figuring out what's important and what's unimportant, what we should pay attention to and what we should ignore. This is to be expected from such a young field and one so different from every other field of study. But we are starting to establish some principles for developing software.

An example of a first principle in software development is the *Single Responsibility Principle*[2] that states "there should never be more than one reason for a class to change."

While this seems like a simple statement, it carries with it a great deal of weight. Since classes act as templates for the objects in a system, it means that we should design our classes so that they represent a single *thing.* This implies a lot. It implies we'll have lots of little classes in the system and each one will be focused on fulfilling a single responsibility.

By narrowly focusing a single responsibility for a class we limit how that responsibility can interact with other classes in the system. This makes it easier to test, find bugs, and extend in the future. The Single Responsibility Principle guides developers to design systems that are well partitioned and modular.

Another example of a first principle in software development is the *Open-Closed Principle* stated by Bertrand Meyer in his book *Object-Oriented Software Construction [Mey97]*, "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

This means we should design systems so they can easily be extended without changing much existing code. When I ask developers why the Open-Closed Principle is important they immediately know why, because changing existing code is often harder and more error-prone than writing new code. When developers understand and value the Open-Closed Principle they tend to write more maintainable code that costs less to extend later.

Principles are very powerful but they're not actionable. Principles tell you what to do but not how to do it. In software, there are many ways to achieve "Open-Closed-ness" in code. It would drive us toward cohesively building objects, programming to abstractions, and keeping behaviors decoupled. It's

2. http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

important to note that these characteristics are side-effects of trying to achieve the principle.

Likewise, there are many ways to achieve the Single Responsibility Principle. Following this principle would drive developers to do things like calling out more of the entities in the problem domain, isolating behaviors, and making the system more modular. All of these things help build more resilient architectures.

Principles often go unstated. Developers may have a vague awareness of what they're striving for, but often don't have it clearly articulated in their own minds. It's helpful to think about principles as the overarching wisdom that helps guide us to do the right things.

To Be a Principle

Principles can be crisp and clearly defined or they can be vague and unarticulated. Whether stated elegantly or not, principles point us in the right direction and take us closer to the true nature of what the principle applies to. Principles may give us insight or just be good advice. And by "us" I mean not just software developers but everyone those developers might come in contact with—everyone on and around a software development team.

Principles help us generalize about a thing. They help us organize our knowledge. Not all principles are equal. Some are purer, more basic than others, and as we saw, principles from which you can infer other principles are called *first principles.*

I think of principles as lofty goals. They're things that we want to strive for because we know they're good and virtuous. We also know that while principles present worthy goals, they're not always achievable. In software, principles represent overarching advice that helps developers build better software.

To Be a Practice

Principles are important but principles alone are not enough. We also need ways of achieving principles in practical situations and that's what practices are for.

I use strict criteria for defining a practice. In order for something to be a practice it must:

- provide value at least most of the time,
- be easy to learn and easy to teach others,

- be simple to do—so simple, in fact, that you can do it *without thinking about it.*

When a practice fulfills these three conditions then it can easily propagate among a team and its benefits compound. You can just *do* the practice and it automatically saves time and effort on an ongoing basis.

The nine practices in this book represent a core set of high value practices that are often misunderstood and misapplied but hold the key to sustainable productivity. And all of them are huge time savers. I'm not about giving developers more work to do. They're already too busy as it is. The practices I advocate developers adopt save them time both in the short term and the long term. They help developers build focused, testable behaviors.

Principles Guide Practices

Developing software is a seemingly endless set of questions and choices. Questioning is a powerful but tiresome process. Should I do one thing or another? Evaluation is important, it's how we come up with new ideas and innovate.

But questioning ourselves all the time is exhausting. I'm not advocating that developers go fully on automatic but it can be helpful to have some general practices that can quickly be applied without thinking about whether you should do them or not and that bring you closer to achieving a principle. For example, the simple practice of eliminating duplication in code can lead to unifying and defining classes, each with a single responsibility, which gets you closer to achieving the Single Responsibility Principle. I've trained myself to quickly spot and eliminate duplication in code. I don't have to ask myself if I should get rid of it, I just do it out of habit, and having that habit makes it easier for me to do good work.

When you understand the purpose behind the practice, all of the practices I'm recommending can be done without too much thought. This gives you tools to better define and build software. Practices replace questions and uncertainty with action.

Principles tell us how to apply practices to maximal effect. They help us use practices to their fullest. Principles are like guiding lights. The help show us how to use our practices correctly.

An example of a principle in investing is "Buy low, sell high." That's really good advice for investing but it's also really crappy advice because I haven't told you *how* to do what I told you to do.

Principles are the things we want to drive toward, and practices tell us how to get there. We can actually *do* practices, so an example of a practice in investing would be dollar cost averaging. If every pay period, or every month—a fixed period of time—I take a certain percentage of my income and invest it, when prices are low I'm buying more stock with that same fixed investment. With the fundamental assumption that the market will continue to go up, I will have "bought low" most of the time. Then if I hold on to those investments until I'm ready to retire, and the price of a share of stock is higher than the average price I've paid over those years of automatic investments, I'll "sell high"—principles (buy low, sell high) and practices (dollar cost averaging) go together.

Practices help us get closer to principles, but principles help us use practices correctly. We get lost unless we keep our eyes on both of them. Either we're great theoreticians but have no clue how to get there, or we're very pragmatic with daily tasks but don't work toward any specific end. We need to balance both.

Each of the nine practices in this book has an important purpose. When you understand the purpose behind the practice you'll know how to apply the practice correctly, when to not apply the practice, and what the alternatives are. Practices apply principles, they are practical embodiments of principles.

Anticipate or Accommodate

Without the right set of practices that support creating changeable code we are unable to easily accommodate change when it happens and we pay a big price. This leaves us in a position where we have to anticipate change before it happens in order to accommodate it later. And this can be stressful.

There's a lot of stress when the team has been working on a big feature thinking, *Okay, it's now all been compiled and oh, boy, I hope it works. I wonder what I'm going to see at the end of this.*

The bottom line is *stress doesn't help build a better product.* And when developers realize that fundamentally the software development industry is built around anticipating change rather than having a time-tested set of principles and practices to accommodate change when it happens, we can see that this still young industry has a long way to go.

That's the dichotomy: anticipate or accommodate, and most developers don't yet know how to accommodate change in software. Imagine if every actor always had to get it right on the first take. Making a movie would be hell, it would be so stressful. Just knowing that you don't have to be perfect on the

first and only pass makes anything so much easier, and by eliminating (or at least dramatically reducing) the stress of performance anxiety, developers find they can pretty much get it right on the first take most of the time, or discover a new idea or way of doing things, and take on more challenges because now they know if they fail the first time they can still recover.

Most of us can't accurately predict the future. What the user might want after the team delivers what was asked for is really anyone's guess. Anticipating future needs can be exhausting and you're probably going to be wrong most of the time anyway. Yet I find a lot of developers engage in anticipating future needs for their code, even though what they're anticipating is not a requirement today. This can cause developers to waste time worrying about features and functionality that are not currently needed and robs them of valuable time to deal with things that are needed right now.

Rather than try to anticipate what the user might want in the future, what if developers could find ways of accommodating change once it's asked for? What if there were a series of principles and practices that developers could follow, without even thinking about it, that would make changing code easier? Then when the inevitable happens and the customer wants a new feature, the code can accommodate it.

This is not just wishful thinking. I believe developers must and can have a series of standards and practices they can share to help deal with change. We'll discover many of these practices together in the rest of this book and with this knowledge you'll be able to discover many others on your own. But before we can look at these practices we must arrive at agreement for what "good" software is so we understand the principle, the reason for using the practice, in the first place.

Defining "Good" in Software

What makes software "good"? When developers look at a design or a piece of code, how do they determine if it's well written? What are the things they look for?

When I ask developers these questions I rarely get a consistent answer. For some people, "good code" must be fast and efficient. For others, it must be easy to read and understand. Still others say it must be bug free.

These are all good things, but how do we achieve them? And when we have to trade one thing for another, where do we draw the line? These can be hard questions that don't often get asked, but they can affect how managers and software developers alike work on a daily basis.