Beyond Legacy Code

Nine Practices to Extend the Life (and Value) of Your Software

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Beyond Legacy Code

Nine Practices to Extend the Life (and Value) of Your Software

David Scott Bernstein

Foreword by Ward Cunningham

edited by Jacquelyn Carter

Beyond Legacy Code

Nine Practices to Extend the Life (and Value) of Your Software

David Scott Bernstein

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Printed in the United States of America.
ISBN-13: 978-1-68050-079-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—June 17, 2015

CHAPTER 5

Practice 1: Say What, Why, and for Whom Before How

Up to fifty percent of development time in a traditional software project is taken up by gathering requirements. It's a losing battle from the start. When teams are focused on gathering requirements they're taking their least seasoned people—business analysts who don't have a technical orientation in terms of coding—and sending them in to talk to customers. Invariably they end up parroting what the customer says.

We all have a tendency to tell people what we think they want to hear, and some customer-facing professionals still fall into this trap. It's uncomfortable to try to find a way to tell your customer, "No, you don't want that, you want this," or "Let's not worry about how it works and instead trust our developers to get us there."

It's easier to list the features requested by the customer and tell them exactly what we think they want to hear: "Got it. We can do that." But can we?

More importantly, *should* we?

It's natural in spoken language to talk in terms of implementation. That's how people tend to speak, and it's hard to identify when we're doing that. I go from *my* specific understanding to a generalization. You hear that generalization and go back to *your* specific understanding. And in the end we have two different experiential understandings of what we think is common ground, but they're probably completely different.

Requirements have no form of validation and that's translated from the customer telling the analyst, the analyst writing it down, the developer reading it and translating it into code… It's the telephone game. There are all these different ways of reinterpreting stuff so when you finally give the released version to the customer they're apt to say, "That's not what I said. That's not what I wanted."

Don't Say How

Customers and customer service managers alike should get away from the whole idea of telling developers *how* to do something, and there are several reasons for that.

The way software is built is not terribly intuitive for people who don't know how to do it. It's not as if everyone can just sit down at a computer and figure it out. So considerable effort is put into trying to translate the software development process into something more understandable to the layman, as a way to help the customer feel more confident that his or her needs are being heard and there's solid agreement as to what the project is and what it will do. That's perfectly reasonable, but unfortunately that effort tends to yield a rigid set of requirements.

Requirements sound like a good idea, but because of the way people tend to communicate in terms of *how* they end up causing more problems then they solve. And by the way, this is a problem in spoken language. It's systematic of the way self-consciousness works and not exclusive to software requirements.

As soon as a team of developers hears or sees a requirement that says *how* to do it, it's as if their hands had been tied behind their backs. It's effectively saying: "Do it this way." And that's what the developers code up, and by default they code it up procedurally rather than stepping back and saying, "Now, how could I create a cacophony of objects to interact and create this behavior?"

Software development is no longer about telling the computer to do this, do that, do this. It's about creating a *world* in which the computer is compelled to do these things based upon the interaction of the objects involved.

That sounds a little like the movie *Tron*, and I don't want to be anthropomorphic about it—obviously there's no consciousness inside a computer—but if you make a hill-object and a ball-object and if you model them well, the ball should roll down the hill.

And something similar should be done with our business rules.

Business rules are the rules of the system, the bits of code inside the *if* statements. Business rules tell the system when to take action.

Software developers want to express these rules in terms of the domain we’re modeling. Whatever domain or business our software is modeling, we want to use *their* terminology and draw on that domain knowledge. We want our business rules to naturally follow from the objects in our software models, or what I refer to as the *problem domain*.

This shouldn’t be too strange to consider. Think about it manually—in terms of real world objects and things. Builders follow a blueprint, clerks organize files in a filing cabinet, but they do those things in a way that makes sense. Builders don’t look at the blueprint upside down or backward or draw blueprints that aren’t to scale, and no one would file the folder marked “Bernstein, David” under P for “person,” or R for “one of the letters in his last name.”

Software developers want to do the same thing with software objects.

The objects in the virtual world of a program should mirror the relevant aspects of the objects in the real world that they represent.

Turn “How” Into “What”

What we should all be going for is a collaborative relationship with the customer. But the problem continues to be the spoken language. It’s simply too easy to fall into a pattern of saying how to do something. And the domain of the how is exclusively the domain of the software developer.

As software developers, we want to know from the Product Owners and customers *what* they want, *why* they want it, and we want to know *who* it’s for—we don’t want them to tell us *how* to do it, because that’s our job. That’s the world software developers live in. We straddle the *what* and the *how*. We’re the only ones who get to know *how*.

If I wanted to have a house built for myself, I’d hire an architect, a contractor, a plumber, an electrician… and I’d tell them I want a kitchen with lots of counter space and a gas range, I want this many bedrooms, a Jacuzzi tub, and so on.

What if the architect then asked me what angle the roof should be, the contractor wanted me to tell him which nails to use, the plumber waited for me to tell him how many feet of pipe to order? My response to all of them would be “Make it according to code, and make it good.” If the architect then asked me what a building code was… I’d get another architect. I don’t know what the code says, but I’m willing—and happy—to hire a trained and experienced professional who does. We trust those codes, even if we don't know what they are, and we trust professionals to do the right thing.

And even if there are no "building codes" for software, there are trained and experienced professionals who can not only handle the "how" but can offer new features, new ideas and approaches, to make the end result something much better than first described.

Software developers are experts in implementation as well as abstraction. In the development process, developers come up with all sorts of alternatives that may not make sense to non-developers, because a good developer's primary concern is maintainability.

And there's far more than one way to do anything.

It's almost impossible for different teams to build software that is exactly the same. I could give a thousand developers the same requirements—and I've done this—and I see a thousand different approaches. The end result may look the same—indeed, they probably *do* look the same—but how they're implemented is different in subtle ways, and sometimes in very big ways. And that's good—that's where new and better ideas come from. But there needs to be some common ground, some standards and practices where the context is set first so the software can be built correctly, without a lot of dependencies on implementation.

As of this writing, I've trained more than 8,000 developers in classes that usually include some form of programming lab. So far about 500 developers have done one of my programming labs that involves writing the guts of an online auction system. No trivial task.

What I've found is a startlingly diverse range of solutions.

Most will have a lot of similarities and they'll generally call out most of the same entities—such as auctions, sellers, and bids—but there'll be subtle differences in implementation. Some will keep a list of logged-in customers, while others simply have the user object keep track of whether it's logged in or not.

There's no right answer for these kinds of things. I don't care how one developer implements a feature, what I care about is that all developers understand the tradeoffs of their decisions and the other paths they didn't take.

Software developers don't have to "standardize" implementation beyond a set of core principles, patterns, and practices developers should be aware of. But they do want to "standardize" how they define behavior, what tests to write.

If developers can agree on what tests to write to specify behaviors then they suddenly arrive at a great deal of common ground.

Have a Product Owner

In every *great* software development project I've ever worked on, we've had a Product Owner. Typical names for this role include product champion, customer representative or onsite customer, project manager, sometimes it's even team manager, or team leader. Whatever you want to call this person, I'll continue to refer to him or her the way Scrum does, as the Product Owner or PO: any individual who is mostly responsible for the product, who's *most* in touch with the customer, and who best understands what the product really is. The Product Owner becomes the authoritative source, and this role is absolutely critical.

Having worked at IBM for many years and built on projects/products designed by committee, I can tell you it absolutely doesn't work.

After all, *most* things designed by a committee don't actually work.

The Product Owner not only guides but drives the development process, though this is *not* necessarily a technical role. In fact, it tends to work better if the PO is not a technical person but really has an intimate knowledge of what the product is—this person is *really* familiar with his or her domain.

And is prepared to take the heat…

On one hand, the PO is a superstar. On the other hand, the buck stops there. The PO is sometimes referred to as *the single wring-able neck.* That person must be the final authority. Even if the Product Owner is wrong sometimes, software developers need to have clear and direct answers to questions. Developers are in the midst of details that most people don't ever think about and few really understand.

The Product Owner is the relationship hub. Everybody goes to that person with updates and questions, and he or she filters that information. The Product Owner holds the vision for the product, defining what is to be built next, though defining *the product itself* is a collaborative effort within the whole team—of course driven by the PO and any subject matter experts (SMEs) with a stake in it.

The Product Owner is the person who says: "This is the next most important feature to build."

The Product Owner orders the backlog, orders the features to be built, ensuring that the most important stuff gets built and the least important doesn't, at least not right away. One of the main benefits of an ordered backlog isn't so much to do the most important stuff first, but to let the least important stuff fall off the edge so we don't waste time on it.

This is the part of our industry that hasn't yet been figured out well.

On a film crew everyone does his job and does it well, and the film couldn't be made without the contributions of each of them. But to make a great story, to make a really great picture, it's the director who knows the flow, the pacing, and everything that makes it come alive. The Product Owner is like a film director—both have to take responsibility for the whole project.

Developers are particularly good at coming up with questions no one has ever considered. They have to ask those questions, in an enormous level of detail, to code software that does what they want it to do. If they don't, if they fail to take into account one possible condition or one potential problem, there is no guarantee what the computer will do, and the program usually crashes.

Sometimes developers get it wrong—not the code, but *what the code is actually supposed to do*—and that's why we need the Product Owner there to answer questions, research the right outcome… even if many times it's not an absolute answer.

We need to carry forward the intent of the user and not get caught up in minutiae or details or politics.

Still, the idea of working without specifications makes some developers skittish. Asking anyone to write software before they have all the requirements, before they *know* what to do, seems inefficient, even reckless. But developers can actually start building without all the requirements, adding as they go, and do it efficiently and with a considerable increase in quality.

This goes back to the idea of asking "What?" not "How?"

So then, what is the alternative to specifications?