EMERGENT DESIGN

The Evolutionary Nature of Professional Software Development

SCOTT L. BAIN

NetObjectives

Product Development Series

Illustrations by Andrea Chartier Bain

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com

This Book Is Safari Enabled

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:

- Go to www.informit.com/onlineedition
- Complete the brief registration form
- Enter the coupon code 8SI6-DAYH-12FU-8VHW-REXK

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Bain, Scott L.
Emergent design : the evolutionary nature of professional software development / Scott L. Bain.
p. cm.
Includes index.
ISBN 978-0-321-50936-9 (hardcover : alk. paper) 1. Computer software—Development. 2. Computer software—Development—Vocational guidance. I. Title.

QA76.76.D47B345 2008
005.1—dc22

2007050637

ISBN-13: 978-0-321-50936-9
ISBN-10: 0-321-50936-6
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, March 2008

Preface

Designing and creating software is hard.

I like that it's hard. I like a challenge. I like solving puzzles. That's probably what attracted me to computers and programming in the first place.

It's just that it's a little bit *too* hard. I don't want it to be easy; I'm not asking for that. I just want it to be a *little* easier, a little more predictable, and a little less chaotic.

I'd like to be able to tell someone, at the beginning of a project, what my software will generally be able to do when it's done, and feel confident that I'm right in what I'm saying. I'd like to be able to tell how long it will take to get the project done, and how much, generally, it will cost. And, I would like to be successful in these predictions and estimates—at least most of the time.

I'd like to feel like I know what I'm doing. Really *know*.

Anyone who has developed any complex software has surely had this experience: at about month 9 of a 12-month project, we're fine; we're on-track. At month 10, we're 4 months behind. How is that possible? Obviously, we were not fine at month 9—we just thought we were. Why didn't we know?

Or, perhaps we have a working system, one that seems just fine, and then the end users want some new function or capability. It is a reasonable request. Things change; we know that. The pace and scope of change in our world is on the rise.

But when we try to make the change the customer wants, things seem to fall apart in unpredictable ways. It makes us hesitant, knowing this can happen. It makes us resistant, even hostile at the prospect of

accommodating such changes. The longer a person has been in development, the more likely he is to feel such resistance.

This is not our fault.

Software development has not been around for very long, in the grand scheme of things. Other, similarly complex endeavors (medicine, the law, architecture, and so on) have been around for hundreds, even thousands, of years, and in that time a whole set of standards, practices, and general wisdom has been captured and handed down from generation to generation. This has helped to increase the rate of predictable success for each new batch of doctors, lawyers, and builders, and in each case has led to the formation of an organism we call *the profession*.

Professions have their own lives, their own existence. For example, the profession of carpentry has been around for thousands of years, though no carpenter is that old. Professions provide a sort of safety net for those individuals in their practice.

The purpose of this book is to examine what we need, as software developers (or *programmers*, if you like), to get that kind of value from what we do, from each other, and from the practice itself. I'd like to take a step back, look at the *nature* of what we're doing, and derive a set of best-practices, general wisdom, and specific patterns of activity that will elevate our business into a true profession, or something akin to that, with all the benefits that such a thing provides.

However, it's not my intention to stay purely theoretical, as interesting as that might be. I want to talk about real things, about the aspects of software development that are too hard, that are too limiting, and to suggest better ways of going about this job. I want to focus on things that are truly valuable.

My contract with you is this: Everything I will investigate, suggest, present, demonstrate, and so on, will have as its core intent the goal of improving our lot as creators of software. No matter how interesting or compelling a thing might be, if there's nothing "in it for us," then I'm going to leave it out.

One thesis I'm going to start off with right now is this: Software development, by its very nature, is a process of evolution. We do not analyze, design, and build; we create something that works, is of high quality, and is valuable as it stands, and then we evolve it in stages toward the product that the world needs. I've got a long way to go to demonstrate this, and in order to get there I'm going to need a set of supportive concepts and techniques.

Here are the things I'll start off examining.

Qualities

How do we know when software is good? Because it works? We all know plenty of software that works but is not good. When presented with two or three ways of doing something, how do we determine which one is best? What does *best* mean? Following the general tenet of this book, *best* should have something to do with value to the developer, and a resulting increase in success, which yields value to the customer. The qualities we will focus on provide this kind of in-the-moment guidance that can help us make better decisions, more reliably: coupling, cohesion, eliminating redundancy, making things testable, and the granddaddy of them all: encapsulation. Included in this discussion will be those negative indicators (pathologies) that can help us to see when one or more of these qualities is not being adhered to.

Principles

What are the fundamental theories that define good software? In other words, what are the points of view we can take on a system that give us a better chance at achieving the qualities, after we know what those are? Principles say "this is better than that" or "this is more important than that." Principles promise better results in terms of the qualities we will emphasize, given that software needs to be able to change in order to meet the needs of a changing world.

Practices

Practices are things that you can do as part of your day-to-day coding activities, which will help you in significant ways. The practices I am most interested in are those that help you in multiple ways, and yet are not a burden. Lots of bang, little bucks. Also, since practices are truly valuable when they are shared and promoted to all the developers on a team (or in an organization or even, perhaps, to the profession), they should be things that are easy to teach others to do.

Disciplines

Similar to practices, disciplines are *things you should do*, but they are larger scale, take longer to learn, and are not without cost. However, the value

they offer is so fundamental and profound as to make them worth the effort and time they require. Unit testing and refactoring are examples of disciplines, as is the notion of test-driven development. I'll cover them all.

Patterns

Patterns represent what we've done before that has worked. But I don't mean just a cookbook or a set of templates; software is more complicated than that. By a *pattern* I mean the set of interrelated points of wisdom that reflect what we, as a group, know about certain situations, those that we find ourselves in again and again. We've been there as a profession, even if some of us have not as individuals. Patterns are a way of sharing the wealth of experience, as a community of colleagues, and supporting one another toward greater success. Patterns are different from principles in that they are *contextual*. Principles apply generally; patterns apply differently in different situations. We'll examine these concepts in terms of each pattern's *forces*, and see how this view of patterns makes them much more useful to us than simply canned designs would be. There are lots of patterns, and lots of patterns books, so I provide an appendix that contains an overview of the patterns I use in the book to illustrate their role in an emergent design.

Processes

In general, how does software development work? How do we find out what we need to build? How do we know when we're done? How do we know when we're on track? And more importantly, how do we know when we're not on track? When we are off track, what do we do? I've tipped my hand already a bit in suggesting that creating software is an evolutionary process, but that's obviously just the seed of the idea.

I'm not alone in this pursuit, of course. In this book, I definitely draw upon the work of others including Alan Shalloway, Martin Fowler, Ward Cunningham, Kent Beck, Ron Jeffries, and Robert Martin, just to name a few. I've learned a great deal from these people and others like them, and I acknowledge their efforts in the Bibliography and point you to the resources they have provided our profession.

I've been accused of being developer-centric, as have some of the colleagues I just mentioned. In my case, this is true. I focus on the developer not just because I am one myself, but also because I believe if we want

better software, we need to do a better job supporting development. To me this means a focus on the developer (e.g., an important part of quality health care is making good doctors). It does not mean that I value software if it does not get used: Unused software is worthless, in my opinion. Therefore, while I certainly focus on those things that help developers succeed, the goal is *better* software and the *right* software, which certainly will benefit all concerned.

There is other work to be done, certainly. I do not pretend to have solved the problem by bringing valuable ideas and practices to my fellow developers; but this is my part along the way.

I believe strongly that software development is on the brink of becoming a profession—in the true sense of the word—and that going that last mile, filling in the missing pieces, is one of the most important activities of our current era. In years to come, I think we will look back at this time and realize that this was the era when software development matured to the degree that it could reliably meet the needs of the modern world, and I'm very excited to be a part of it.

So, let's begin.

CHAPTER 3

The Nature of Software Development

To do something well, it helps to understand it. I think we all want to do a good job as software developers, and I think we will have a better chance of doing a good job if we take a little time to investigate the nature of what we are doing.

Once upon a time, I was a Boy Scout. My troop was planning a white-water rafting trip down the Colorado River in the eastern part of California, just above the Imperial Dam. We were given safety training before the trip began, and one thing they emphasized was what to do if we fell out of the raft, which was a distinct possibility in some of the rougher sections.

They said, "If you become separated from the raft:

1. Keep your life vest on, and
2. Do not fight the current."

Keeping your life vest on means you will be less dense than the water (which is why you float), and not fighting the current means you will not run into the rocks. The current, after all, goes *around* the rocks, not through them, and as you will be less dense than the water, it will be able to take you around them too.

This can be uncomfortable. The water can go through narrow defiles that are not fun for a person to go through, but it will not slam you into a boulder that can injure you or kill you. So, do not fight it. Let the nature of the river take you.

This lesson has many implications in life, but here I am suggesting that software development has a nature, a way it "wants" to flow. If we follow this by coming to understand it and then align ourselves with it, then everything will get easier, less dangerous, and ultimately more successful.

I'll begin with the notion that software projects fail more frequently than they ought to, and examine why this might be. I'll examine some of the fundamental assumptions that underlie the traditional view of the nature of what we do, and then look for ways to change these assumptions in the light of what we now have come to understand about software and its role in the world. Finally, I'll show how this new understanding will impact the choices we make, the qualities we emphasize, and how design patterns can help us to accommodate change in systems.

We Fail Too Much

For most of my adult life and some of my teenage years, I have been writing programs. I started using Basic and Pascal and then moved on to C and other languages, including such esoteric and unusual examples as PL1 and Forth.

Moving from language to language, platform to platform, era to era, much has changed. But there are some fundamentals that seem to have held their ground. We have looked at some of them (side effects, for instance, always seem to be a concern).

One of these constants, perhaps the one that has stuck in my throat the sharpest, is the notion that the failure rate of software projects is way, way too high.

Of course, for the majority of my career I was never sure if this was just my own experience; maybe I was just not very good at this stuff. I would ask other people, friends, and colleagues who also "cut code" and they would tell me their war stories, projects they had been on that had gone disastrously wrong.

But, I also knew that people like to tell those kinds of stories. It is human nature; war stories are dramatic, exciting, and often lead others to try and "top" them in the sort of round-robin, coffee-klatch conversations that developers often get into.

So, was it just me, this low rate of success? What did I even mean by success, anyway?

I was pretty much responding to a feeling: I did not have the confidence that I would like to have; I did not feel like I had a "handle" on things a lot of the time.

Definitions of Success

What defines success in software development? In summary, a software development project is successful if it delivers the value expected for no more than the cost expected. This means

- The software is ready on time.
- Creating the software cost what it was supposed to cost.
- The software does what it needs to do.
- The software is not crippled by bugs.
- The software gets used, and does make a positive impact. It is *valuable*.

The first two, of course, are often related to each other. The time we spend to make software is a big part of the cost, because the largest cost is developer time in the vast majority of projects.

However, I have been on projects that delivered on time, but required the efforts of a lot more developers than was planned for, so the project ended up costing more. Either way, when we are late or over budget, we harm the business we are trying to help; it often has plans contingent on the release of the software, and having to change those plans can be expensive and sometimes harm its relative competitive position in the marketplace.

The third point, which could be called *functional completeness*, is of course a relative thing. Software could always "do more," but the real question is whether the software does the most critical things it was needed to do, and therefore has a potential for a positive impact that is commensurate with the cost and effort it took to make it.

No software is free of bugs but there is a clear difference between software that fundamentally works, with a bug here or there that will have to be remediated when found, and software that is so buggy as to be unusable.

Sometimes software that is delivered "on time, on budget" is not actually finished when it is shipped, which becomes clear when we find a lot of bugs, or find that a lot of crucial features are missing.

In the final analysis, the real critical issue is: Are they using it? How much value is it producing in the world? I think that sometimes we have taken the attitude that this is not in our scope, not our problem. We made it, it does what we promised, and it works, so if they do not use it, then that is their problem.

I do not feel that way anymore. I don't think we can.

First, if the software is not being used, why is that? Perhaps the problem it was intended to solve has disappeared, or changed so much that the software no longer addresses it well enough. Perhaps the customer was wrong when he asked for features x, y, and z, and found out too late for me to change what I was making. Perhaps the software works, but is too hard or unpleasant to use, and people shy away from it.

Whatever the reason, software that sits on the shelf, unused, has no value (see "Appendix C, The Principle of the Useful Illusion," for my thoughts on this). In a sense, software that exists only as little reflective dots on a CD-ROM and is never running, does not really exist at all in any useful way.

I do not want to spend my time on valueless things that do not really exist, do you? I think one of the attractive things about software development, at least to many of us, is the idea that we can *make things* and that these things actually work (and do something useful). That an idea, something that starts in my mind, can have a life outside of my personal domain is a little bit magical, and it is what got me hooked on computers and computing in the first place.

Also, I want to go on doing this. I want to have lots of opportunities, and to be able to choose from a wide variety of projects, choosing the one(s) that are most interesting to me. If software does not deliver sufficient value to the people who pay for it, I don't think this is likely to happen.

So, if we can agree that this checklist is right (or at least close to right), the question remains: How are we doing? Luckily, someone took on the task of answering this question for us.

The Standish Group

In 1994, a think tank known as The Standish Group[1] set out to answer this question. For more than 10 years, it studied a whopping 35,000 development projects and evaluated them in a number of ways. Its definition of *success* was nearly identical to the five-point checklist I outlined earlier.

The report Standish produced is long and detailed, but the overall, bottom-line numbers were shocking and, well, believable to anyone who has ever been involved in making software.

1. http://www.standishgroup.com

Projects that "hit all the marks" were called *successful*, of course. Those that were utter disasters, those that used up time, resources, and energy but were ultimately cancelled without producing anything were called *failed*. And, of course, there are many, many projects that end up with something, but are late, too expensive, very buggy, and so on; these were called *challenged*. Only 16% of the projects were classified as successful! The majority of projects, 64%, were classified as challenged, and 20% were considered failed.

Troubling.

First of all, as a teacher, I can tell you that very few developers, even today, know about these numbers. On the other hand, people who pay the bills for software development know them very well.

They know them and they sometimes think they can explain them. The interpretation I hear most often from project stakeholders when they quote these numbers is that software development is not conducted in a predictable way and that the processes that control it are too chaotic. Some even go so far as to say that software is not really developed in a professional manner, that software developers do not really focus on delivering business value; they just want to be involved with the cool tools and work around neat-o computers. I have even heard some business people express the opinion that developers actually *like* to deliver buggy code so that they can be paid to fix the bugs. Either way, the image that a lot of nontechnical business people have about software developers is basically that of the closeted nerd: unsocial, arrogant, and unconcerned with the needs of "normal" people.

The failure rate suggested by The Standish Group study would *seem* to fit this view.

I travel and teach all over the country and overseas as well. I have met more software developers in the last six months than you will likely meet in your whole life. The clear impression I get from meeting so many developers in so many parts of the world is that the notion that we, as a group, do not care about doing valuable work is *dead wrong*.

I admit that I used to know a few guys like that, back in the 70s and early 80s. There were developers who would overestimate their projects on purpose, and spend most of the time playing games or posting on bulletin boards. The image of the software developer as basically the comic book guy from *The Simpsons* was a part of the mythology of the time.

But those guys—if they ever did exist—are pretty much gone. Sadly, the image continues to live on. So, if you feel from time to time as though you

are not trusted by the people who pay you to make their software, there might be some truth to that.

Besides the waste involved, there is another reason these numbers are so disturbing. I suspect that those projects called *challenged* and *failed* earlier, which in the Standish study amounted to 84% of all projects, are quite often those "death march" experiences that burn out developers.

We cannot afford the high turnover that has become commonplace in the software development industry, but it is unrealistic to think that your average adult developer will stay long in the business after he or she has gone through two, three, four, or five projects that required late nights, weekends, hostile and panicked managers, and an exhausting lifestyle that is hard to maintain.

Also, universities are seeing notable declines in enrollment in their computer science and computer engineering programs. It seems that the next generation does not view our business as all that attractive, and part of that might be the impression that the death march is the normal way software is made.

So, one way that this book could potentially be valuable to you would be if I could suggest a way to improve on these numbers. If we were more successful, this would have at least three positive effects.

- Software would be developed in a more effective/efficient manner.
- The people who control the purse strings of development would trust us more, would treat us more like professionals, and would tend to be less controlling.
- We would be able to sustain long-term development careers more reliably.

I am going to try to do that, but first we have to examine another troubling piece of data from that same study.

Doing the Wrong Things

Another rather stunning figure from The Standish Group study concerned the number of features in the average piece of software that actually end up getting used. Figure 3.1 summarizes the findings.

Yes, you are reading that correctly. It says that 45% of the features we, as an industry, put into software are *never* used by anyone. Never.

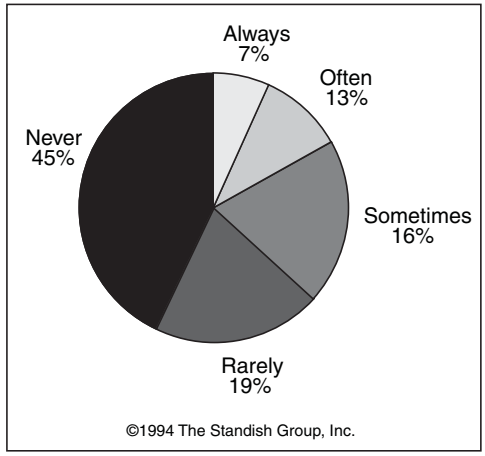

Figure 3.1 Breakdown of features by usage

If that does not bother you overly, think of it this way: That 45% represents our time—time to analyze, design, code, test, debug, redesign, re-code, re-test, and then finally deploy features that no one wants, no one needs, and no one will use. How much better could you do your job if you had 45% of your time back? Furthermore, this means that 45% of the code we are writing represents needless complexity. We are adding classes, tests, relationships, and so on that are never needed and are almost certainly in our way when we are maintaining the rest of the system.

Also, take note: The percentage of use is consistently smaller as features are used more often. Rarely is 19%, which is less than Sometimes, which is 16%, and so on down to Always, which is a mere 7%.

We get this wrong, not only at the extremes of Never and Always, but at *every point in between*.

Why? What is it about our process that drives us to make the wrong things, more often than not? If we could figure out a way to stop doing that, or even just a portion of that, this alone could improve our lot significantly.

Doing the Things Wrong

I teach quite a lot. This gives me access to lots of developers, day in and day out, and so I like to ask them questions about how things are going for them, and why they think things are the way they are.

In my most popular class, which is about patterns, I begin by throwing a question out to the entire class:

"How many of you have been on a failed project, or one that was significantly challenged?"

Naturally, they all raise their hands.[2]

Next, I ask:

"What happened?"

This is a question they do not expect, and one that likely they have not been asked much before. Sometimes, a largely unasked question leads to very valuable discussions. I listen and write their responses on the whiteboard. Here are some typical responses.

- The requirements changed after we were pretty far into developing the code.
- My project was dependant on another project, but we did not get what we expected from the other team, and so we had to change our stuff to match what they did.
- The customer changed their minds. A feature they were sure they did not need ended up being important, because the marketplace changed while we were developing the software.
- We did not realize the best way to get a key feature to work until we got well into the project.
- The technology we were using was unreliable, and so we had to change horses midstream.
- The stakeholders imposed an unreasonable schedule on us.
- We committed to a schedule we could not meet, as we later determined.

. . . and so on. I have gone through this process dozens of times, in large and small companies, with individual developers in public courses, here in the United States and abroad, and the answers are basically always the same.

---

2. I used to start by asking how many have been on a completely successful project. The fact that almost nobody ever raised their hands was too depressing, especially to them, so I stopped asking that one.

Looking for a through line, a common element that seems to underlie all or most of these causes of failure, consistently drives to the same conclusion. There is a thing that makes us fail, and that thing is *change.* Requirements change.

- Our customers do not know what they want; or
- They know what they want but they don't know how to explain it; or
- They know and can explain what they want, but we misunderstand them; or
- They know, can explain, and we get it, but we miss what is important; or
- All of this goes right, and the marketplace changes around us anyway; and so things have to change.

Technology changes. Fast. Faster all the time. I do not think there is much point in hoping that this is going to stop, that we're going to finally have a stable environment in which to develop our software.

Change has been, traditionally, our enemy, and I think we have known it for a long time.

Who would rather work on maintaining an old legacy system, as opposed to writing something new? I ask that question a lot too, and rare is the developer who would choose maintenance.

Maintenance is, of course, change. Change is a chance to fail. Nobody wants to be the one who broke the system. Nobody wants to be the one who's going to have to stay late, or come in this weekend, and fix the thing.

And so, we find ways to avoid making changes.

Sometimes, we put a contract or a control board between ourselves and our customers. Sign the contract, and they cannot change the spec unless they jump through a series of hurdles, which are designed to discourage them from making the change at all.

But even if that works, it does not work. Remember that fifth metric of software success? The software has to be used, it has to deliver value. If it does the wrong things, even if they were the things in the signed contract, then it will fail in this respect.

Another approach is to try to create a design that is set up to change in any possible, imaginable way.

But even if that works, it does not work. This is typically called *over-design* and it creates a whole other problem: systems that are bloated at best and incomprehensible at the worst. Plus, my experience is that the one thing that comes along to derail you later is the one thing you didn't think of in the over-design—Murphy's Law, as they say.

So, what do we do? How can we safely accommodate the changes our customers want, and likely need? What has made us so brittle?

Why did we ever try to develop software the way we did (and the way some people still do)? Examining this question can lead us to some interesting conclusions, and can perhaps give us some guidance that will help us to continue this upward trend, and perhaps even speed things up.

Time Goes By, Things Improve

The initial findings quoted in the preceding section are from the first Chaos Report, which was issued in 1994. Since then, Standish has updated these findings each year, basically taking on a year's worth of data and allowing the oldest data to fall out.

We are doing better; that's the good news. The failure rate, for instance, had dropped from 20% to 16% by 2004, and the rate of success had almost doubled, up to 34%. Taken together, failed and challenged projects still dominate, but the trend is definitely taking us in the right direction.

Jim Johnson, the chairman of the Standish Group, has said that a lot of this improvement has come from the trend toward developing software in smaller pieces and from a renovation in the ways projects are managed.[3] That makes sense to me. However, even in 2004, according to Standish, we still wasted over $50 billion in the United States alone. We're on the right track, but we have a lot farther to go.

One Reason: The Civil Engineering Analogy

Back in the day when microcomputers[4] first hit businesses, they arrived on the scene (in many cases) without a lot of planning. It was hard to

3. http://www.softwaremag.com/L.cfm?Doc=newsletter/2004-01-15/Standish
4. They were not even called PCs yet because they were relatively rare.

predict what impact small, interactive systems would have on business processes, productivity, the nature of the workday, and so on.

At the same time, a lot of software developers were ahead of the curve on this; very early micros (like the Apple II, the Heathkit, and the TRS-80) were sold more to hobbyists and experimenters than businesspeople. Naturally, these were often the people who were later called on to write the software for these new PCs that started showing up in the workplace.

That was how I started, in broadcast television and radio; making software was not my "job," but people had heard that I had built my own computer and knew how to "program," and would ask me to write things for them since, at that time, little software existed for the PC, especially the vertical-market software that would apply specifically to broadcasting. Many of my friends and colleagues were experiencing the same things in banking, insurance, education, and other industries.

In those days, software *seemed* to sort of "happen," but it was extremely difficult to predict when it would be ready, or what it would do, or what it would be like. This was probably annoying to stakeholders, but acceptable since software was only a fringe issue for most businesses.

All that changed in the 80s with the introduction of inexpensive clone computers.

In a remarkably short few years, PCs were everywhere, and the vast majority of business processes, including very critical ones, had moved to the PC and therefore to software. It was rather shocking how fast this happened, and most people did not realize it had until it was already a done deal.

Business people suddenly realized how very vulnerable they were to the software used to run their business. The software's quality, when new software would become available, what it did and did not do, and so forth, had a profound impact on a company's well-being.

In the mid 90s, the CBS Network hosted a new media conference in Chicago, where the primary topics included the impact of the Internet on broadcasting, and just exactly how the affiliate stations could get control over their exploding automation. Most of the people in attendance were station managers; I was one of the very few techies there, and as such was a popular person to have at your dinner table. These guys were pretty worried.

They knew, and I knew, that the development of software had to be brought under some form of management, that developers had to be

accountable to and supported by some kind of process. The chaotic style of developing software was producing chaos in business, and this had to stop.

Business people also knew that they had *no idea* how software development worked, or how one could control it. They also did not consider the making of software to be a professional activity, so they didn't ask us (not that we would have known what to say if they had).

They looked around at the other industries they *did* understand to try and find an analog or metaphor that could guide them. A lot of different things were tried, but what was settled on was what came to be called *the waterfall*.

The idea was that software is essentially like building a large, one-off structure, like a bridge or a skyscraper. The project is complex, expensive, and will be used by many people. We never build the same thing twice, but there is a lot of repetition from one edifice to another.

We all know this process, or something like it:

1. Analyze the problem until you "know it," and then document this.
2. Give this analysis to designers who will figure out how to solve this problem, with the given constraints, using software, and then document this design.
3. Give the design to the development team, who will write the code.
4. Hand it off to testers to do the quality assurance.
5. Release the code.

This has been remarkably unsuccessful, but you can understand the thinking.

Before we build a bridge, we analyze the soils, find the bedrock, measure the high and low point of the river, figure out how many cars it will have to handle per hour at peak times, research local laws, see how the bridge will interact with existing roads and structures, measure if ships can clear it beneath, and so on. Then we get an architect to design a bridge that meets these requirements.

If he does the job correctly, the construction team's job is simply to follow this design as accurately and efficiently as possible. Many of the engineering disasters that have become famous in recent history can be traced back to changes, seemingly insignificant, that were made by the construction

team to the plan prepared by the engineers. Because of this, the virtues of construction are considered to be *compliance* and *accuracy*, not invention.

The tester (the building inspector) comes last, to make sure the bridge is safe, work up a punch list for changes, and then finally approve it for use. Finally, the cars roll onto it.

The problem is that analysis for software development is not like analysis in engineering.

When we analyze, we are analyzing the requirements of a human business process, in which very few things are fixed. To communicate what we find, we do not have the highly precise languages of geology, law, hydrodynamics, and so forth to describe the situation; we have the relativistic language of business processes, which does not map well to technology. Thus, communication is *lossy*.

And even when we get it right, business processes and needs change, as do our perceptions as to what was needed in the first place. Bedrock *is* where it *is*, and will *still* be there tomorrow, but people change their minds about how they should run their businesses all the time.

They have to. The market changes around them. In fact, the realities, priorities, and constraints of business are changing at a faster rate every year. Also, the variations in software systems are generally much greater than the variations encountered when building the 247th overpass on Interstate 5.

Also, the analog to the construction step in civil engineering would seem to be the coding step in software development. I think that is a misunderstanding of what we really do when we code. If the virtue of construction is compliance and accuracy, I would equate that to the compile process, not the coding process. If so, where does the creation of code fit? Analysis? Design? Testing? This would seem to be a pretty important distinction to make, and yet we have traditionally left it in the construction position.

So, given our failure rate, why have we continued to develop software like this?

Giving Up Hope

Of course, one reason we have not changed the way we do things is because most of us have assumed the problem is not the process, but ourselves. In other words, we have believed that the reason the waterfall process does not succeed very often is that we are not doing it right.

Software developers have often been thought of as arrogant. I disagree; I think most are very self-questioning and tend to lack confidence. This can sometimes come off as arrogance, but in general, self-assured people don't have to brag.

Lacking any defined standards, software developers have traditionally had no way to determine, for themselves, if they were really *good* at their jobs. Without any notion of what makes a good developer, we live and die by our latest success or failure. This lack of a set of standards is part of what has kept us from becoming the profession we should be, in my opinion.

So, because we tend to worry that the failures in our projects stem from our own faults, we do not suspect that the problem might be the methodology itself.

Einstein said it: "Insanity is doing the same thing over and over again and expecting a different result."

I think we have been a little bit nuts. Well, maybe not nuts. Maybe just hopeful.

Think back to your high school mythology class and the story of Pandora's Box. Pandora, the myth goes, opened a box and unwittingly released all the evils of the world. The last thing that emerged from the box was hope.

When I was a kid, upon hearing this tale, I thought "Awwwwww. Well, at least we have hope!"

No, no. That is not the point. Instead, the point is that *hope is evil*.

As long as we hope that *this time* we will get the requirements right, and stable, then we will keep doing things the way we always have. If we hold out the hope that *this time* we will be able to write code without bugs, then we'll keep writing it as we always have. The hope that *this time* we will be on time, be on budget, just be *better* will keep us from changing what we do.

We give out T-shirts at our courses with cartoons and pithy phrases on them, and by far my favorite shirt says:

"I feel so much better since I gave up hope."

Ignoring Your Mother

Another reason that many of us probably stuck with the waterfall is that it seems to follow the advice our mothers gave us: "Never put off until tomorrow what you can do today."

Waiting is procrastinating, and procrastination is bad, right?

Well, not always. I did some work for a company that created security systems for AS400 computers in Java. When I started that project I did not know anything (at all) about AS400s, only a modicum about security, and only the aspects of Java that I had used before—certainly nothing about the nuances of Java on the AS400.

After I had been on the project for a few months, I knew a *lot* about AS400s, a *lot* about security, especially as it applies to that particular operating system (OS400), and a *lot* about Java as it is used in that environment.

Anyone who has coded anything significantly complex will tell you that midway though the coding she became *much* more knowledgeable about the business she was automating, whatever it was.

Spend six weeks writing code for an embroidery machine and I guarantee you will know more about embroidery afterward than you ever knew (or wanted to know) before. Why not capitalize on this fact, and let what we learn (later) influence and *improve* the design?

Mom said, "Do not put things off." She was right about that when it comes to some things, maybe most things, but not here. I should put off everything I can, because I will be smarter tomorrow than I am today, in just about every way.

Of course, the key to making that work lies in knowing what you *can* put off and what you *cannot*.

Bridges Are Hard, Software Is Soft

There are advantages to the "softness" of software that the waterfall-style approach to software development does not allow us to capitalize on.

After you have laid the foundation for a skyscraper and put up a dozen floors, it is pretty hard to move it one block to the north. That is the nature of physical construction, and that is why architectural plans are generally not made to be changed; the cost of change is usually too high, or even impossible.

In software, however, there are ways of designing that allow for change, even unforeseen change, without excessive costs. Those ways are a big part of what this book is about.

We Swim in an Ocean of Change

One crucial aspect of success for any endeavor is to understand and react to the environment around us. When life gives you lemons, you make

lemonade. You can try to make iced tea, but you are always going to be struggling to make it taste right.

It may well be that a process like the waterfall was appropriate at some point in the history of computers. Those who worked in data processing, for example, did tend to follow a very similar process, with a reasonable degree of success.

But the pace of change, the nature of change, and the ubiquity of change has altered the forces driving our success. The pace of change in business is on the rise. The pace of change in technology is on the rise. People change jobs, teams, and organizations much more frequently than ever before. The Internet has driven commercial change to a much higher rate than ever before.

And there is more to come. Computers, technology, and therefore software are permeating every aspect of our lives, and as we move faster as a species as a result of technology, technology will have to move faster as well.

Accept Change

So, give up hope and ignore your mother.

The first step to making a positive change is realizing that what you are doing now does not work. Requirements, technologies, priorities, teams, companies, everything will change. (See Figure 3.2.) There are things you cannot know until you know them. Give up.

Figure 3.2 The futility of fighting change

Now, if we accept these things, we can ask ourselves this question: Given all that, what would a really powerful way to make software look like? That is a good question, the right question, and it is the question about which many of our colleagues today are concerning themselves.

I am not sure we know the answer yet, but I think we are getting close, and we are certainly a *lot* closer to answering it than we would be if we had never asked.

Change is inevitable. Our process must be a process, therefore, of change.

Embrace Change

What is being suggested now, first by the fellows who dreamed up eXtreme Programming (XP), and then later by the entire *agile* movement,[5] is that we work in relatively short cycles (two weeks, a month, etc . . .), and that in every cycle we do every step—analysis, design, testing, coding—and then let the customer look at what we have done and tell us if we are on track or not.

Each cycle allows us to adapt our design to what the customer has told us, and also capitalize on what we have learned about his domain by working on the system in detail.

Part of what makes this work is *time boxing*, the idea that we limit the length of time we will work on a system before we stop and get what we have done so far validated (again, read Appendix C for my views on this).

The value of this is manifold, but right off we can see that it will prevent us from trying to predict too much. If *validating* the software is part of *making* the software, we will not try to gaze into our crystal ball and predict what we'll be putting into the iteration two months down the line; we will wait until we see how *this* iteration went.

It is also an acknowledgment that these phases (analysis, design, code, test) are highly interrelated.

- Your design is a reflection of your problem domain—while determining the design, insights on the problem domain, and therefore requirements, arise.
- Coding gives you feedback on the practicality of your design, and also teaches you about the problem domain.

5. http://www.agilealliance.org/home

- Considering the testability of a design can help you evaluate its quality (more on this later).
- Often, we come up with questions while designing, coding, or especially testing, which inform our analysis of the domain.

In other words, working incrementally changes "I wish I had thought of this sooner" to "Now I get it; let's do it this way." It is a fundamentally stronger position to be in.

Capitalize on Change

Agile processes are, among other things, an acknowledgment of the realization that software development is like . . . software development. It is not fundamentally like any other activity.

We are in the midst of defining a process that is uniquely suited to the nature of our particular beast, which should go a long way toward making us more successful.

It is also, I submit, another aspect of software development evolving into a true profession. We do not make software the way anyone makes anything else. We have our own way, because what we do is unique, complex, and requires special knowledge and skills.

Maybe that is part of what makes something a profession. Doctors do not work like carpenters. Lawyers do not work like teachers. Complex, unique activities tend to find their own process, one that suits them uniquely.

That said, it can be tricky to code in an agile process, where change is not only allowed for, but expected (and happens more frequently). If we ask the customer every couple of weeks if we are on track, there will generally be things that are not quite right, and so the answer will rarely be "Yes, that is fine." Since we are asking for validation more frequently, we will be making changes more frequently.

Frankly, even if we *do* get it perfect, it is human nature for the person being asked to validate the software to find *something* wrong. After all, he needs to justify his existence.

Earlier, I wrote that "there are ways of designing that allow for change, even unforeseen change, without excessive costs." I wonder how many of you let me get away with that.

"There are ways" is pretty vague, which is an important issue if, again, we are going to allow the customer (and ourselves) to change things more frequently.

Fortunately, this book is also about how patterns promote professionalism, and the value they bring to an evolutionary process. Pick any pattern you like and whatever else it does, I guarantee that it makes your design easier to change. Here are a few examples that should be familiar to you if you have studied design patterns already. (If you have not yet studied them, I have included in Appendix B, "Overview of Patterns Used in the Examples," descriptions of all of the patterns I use in this book.)

- *Strategy.* The Strategy pattern enables the programmer to substitute an algorithm or business rule without affecting the code using the rule. This makes it easier to add new implementations of an algorithm after the design is in place.

  For example, if you are amortizing the value of fixed assets and the customer says, "Oh, by the way," in the final month of your project, telling you about some other method of amortization he will need, you can plug it in more easily if you used a Strategy to vary it in the first place.

- *Decorator.* The Decorator pattern is designed to enable you to add functionality in front of (or after) the main entity being used, without changing the entities that use it. This makes it easier to add new functions, in different combinations and with varying numbers of steps, after the design is in place.

  For example, if your customer has reports with headers and footers, and then suddenly remembers a special case where an additional header and two additional footers are needed, you can add the functionality without changing what you already created, if you used a Decorator to encapsulate the structure of the reports, and the number and order of the headers and footers in the first place.

- *Abstract Factory.* The Abstract Factory pattern controls the instantiation of sets of related objects used under particular circumstances. If you design your main code so that it can ignore the particular implementations present, it can let the Abstract Factory decide which particular objects to use. This makes it easier to accommodate an entire new case.

> For example, if you design the system to run on UNIX and NT, and the customer realizes that, in this one case he forgot about, it needs to be deployed on Solaris, an Abstract Factory would allow you to add an entire new set of device drivers, without changing anything else in the codebase.

Of course, if you are *not* already up to speed on these patterns, fear not. We will dig into them more fully later in the book.[6] Simply know that they are not just cool ideas or clever bits of code; they are part of the fabric of our profession.

Patterns were and are discovered, not invented. Something becomes a pattern because it worked, because it helped someone else achieve a better design at some point in the past, and because that person recorded this success and passed it on to you—just like carpenters, masons, and lawyers have been doing for centuries.

And, just like *their* patterns helped them create their own unique processes for *their* fundamental professional activities, so do ours.

- Patterns support agility, even though patterns were originally discovered before agility became a force in our industry. Agility, in turn, helps codify a unique process for software development, and this helps define a professional community.
- Patterns help us to communicate, and to define what we are doing within the unique and specific boundaries of the software development profession. Moreover, the communication that patterns enable is not merely implementation-speak, but captures nuance, wisdom, caveats, and opportunities.
- Patterns provide a clear path to entry for new developers, who are seeking the repository of knowledge from the profession they aspire to engage. Studying patterns is a clear way to strengthen your grasp of good design.

This is why, at Net Objectives, we teach patterns in this way. Not as reusable solutions but instead as professional best-practices. How a pattern is implemented depends on the specifics of the problem domain, the

---

6. Others have written good books on the subject (see the Bibliography at the end of the book). Also, we've created a pattern repository for collecting this information (see Appendix B, "Overview of Patterns Used in the Examples").

language, and framework being used, and so on but the value of the pattern is always constant. This turns the study of patterns into a much more valuable pursuit.

Once patterns are understood at this level, however, a new and more interesting way to think about them emerges. In Alan Shalloway and James Trott's book, *Design Patterns Explained: A New Perspective on Object-Oriented Design*, they discuss

- The principles exemplified by patterns
- The practices suggested by patterns
- A way to apply these principles and strategies in software development generally, even when a particular pattern is not present

This new way of thinking about patterns moves us even closer to becoming a profession as we identify universal principles that inform software development. As patterns become more prevalent, a greater sharing of terminology, thought processes, concerns, and approaches becomes more widespread.

A Better Analogy: Evolving Systems

What is the nature of software development? Once upon a time, someone said,

"All software systems decay over time to the point that, eventually, replacing them is less costly than maintaining them."

This was stated as an inevitability, and I think we have tended to accept it as the truth (see Figure 3.3). Well, I may be embracing change, but I am rejecting this "truth."

If change is inevitable, I would agree that decay is inevitable only if change must always be decay. I think it has tended to be like this because we, as an industry, have not focused ourselves on making code inherently changeable. Our tradition, even if you take it back just a few years, is based on the idea that

- There is relatively little software in the world.
- Computers are very slow, so performance is the main issue.
- Memory and storage are expensive, and the technology is limited.
- Computers are more expensive than developers.

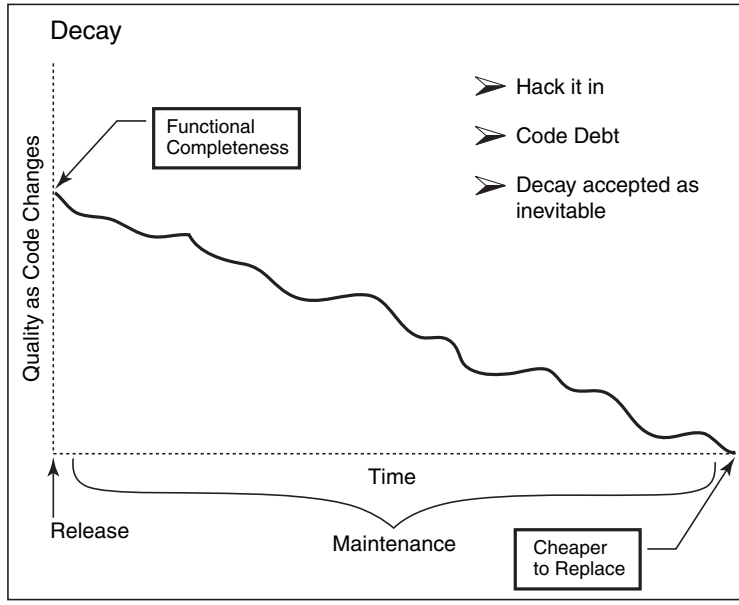

Figure 3.3 An inevitable truth?

If you make it faster, you are on the right track, right? Once upon a time, perhaps, when performance constraints were often make-or-break. But today?

- Computers are now a *lot* faster than they were then (orders of magnitude).
- Memory and storage are cheap and getting cheaper, vast and getting vaster.
- We have a *lot* more software to deal with.
- Developer time is a major expense in software development, maybe the main expense.

Computers are only going to get faster and more pervasive in our lives. And software is a *lot* more important today than ever in the past. Go get an MRI: Your doctor will be using software to find out what is wrong with you. Call 911 on your cell phone: Software sends the police to your aid. And tomorrow, who knows what software will be doing for us?

Look again at Figure 3.3. Do you want to fly in an aircraft that is primarily controlled by software that has decayed to the point where it is really bad, but not quite bad enough to throw away yet? No? Neither do I. That is why "hack it in," which was how we used to make changes, will not hold water any more.

Ward Cunningham of the XP Group says that when you make this kind of change you are incurring *code debt*. It is like putting something on a credit card because you cannot afford to pay for it right now.

Will it cost more or less later on? If it is hard to fix the problem properly now, will it be easier in a few weeks when the code is no longer fresh in your mind, and when the hack you put in today has caused three other, interdependent problems to emerge?

What if I did the right thing now? The argument against this, of course, is that it will cost more to do things right. I am not sure I buy that, but even if I did, I would say that the cost spent now is an investment, as opposed to a debt. Both things cost, but investments pay off in the future.

So I favor the approach shown in Figure 3.4.

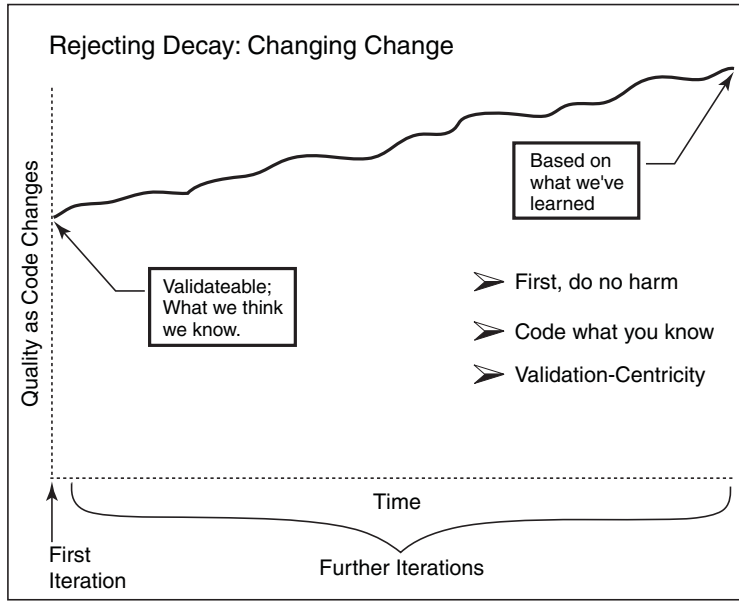

Figure 3.4 Rejecting decay: changing change

I say that decay is not inevitable if we refuse to accept it.
For this to be true, we must assert several things.

- We need something like the Hippocratic Oath: First, do no harm. I do not expect my software to be perfect (ever), but I think it is reasonable to hold myself to the basic standard that every time I touch it, I will take care not to make it any *worse*.
- We need to center ourselves on the notion that validating software (with those who will use it) is part of making the software.
- We need to code in a style that allows us to follow this do no harm oath.

So, what is that style? What is harm? What is the nature of maintainable code? How can we accommodate change if we are accepting that we don't know what/where it is when we start our project?

If we can answer these questions, we can be responsive. We can change our code in response to what we learn, when we learn it. We can keep the quality high, allow the customers to ask for new features when their business changes, and watch the system get more and more and more appropriate to the problem it is designed to solve.

My word for this is evolution. I think it is the essential nature of software development to be evolutionary, that this has always been so, and that the failure and pain in our industry comes from trying to make it all work some other way, like science or engineering or manufacturing or whatever.

Summary

I attempted in this chapter to establish that the nature of the software development is best characterized as one of evolution. The purpose of this book is to examine this notion of software development as evolutionary by nature, and then to enable it, support it, and give you the power that comes from developing in this way. If you agree, read on.

Evolution Versus Natural Selection

I know I am putting my head in the lion's mouth when I use a word like *evolution*. But I do not want to re-try the Scopes Monkey trial again. I am not talking about natural selection and the origin of the species here.

Evolution is a more general concept, and here I am talking about a conscious process of change, something we as humans do on purpose.

Therefore, I do ***not*** mean

- *Random mutation.* I do not mean we should just try any old thing until something arises as workable. We always do the best we can; we just give up hope on certainty.
- *Natural Selection.* I do not expect this just to happen on its own.
- *Millions of years.* Schedules tend to be a little tighter than this.

By evolution, I ***do*** mean

- Incremental, continuous change.
- The reason for the change is *pressure* from the domain.
- This pressure is a positive force. (We finally understand a requirement, or the requirements change in a way that will help solve the problem better, and so on.)
- That our response to the pressure is to change the system in a way that makes it more appropriate for the problem at hand.

This makes change a positive thing, because change means improvement, not decay. It means we have to be more than *willing* to change our code; we have to be *eager*. That is a big shift for most of us.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

Q

R

S

T

U

V

W

Y–Z