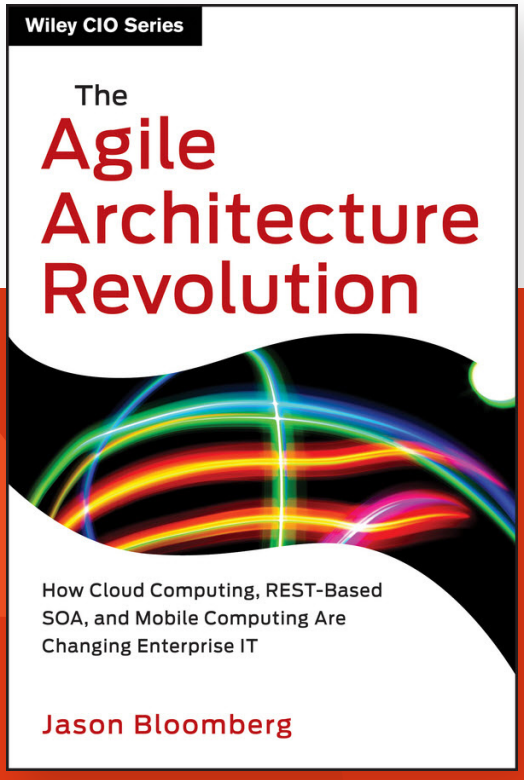
YOUR COMPLIMENTARY CHAPTER



CHAPTER 1:
Introducing Agile Architecture

The Agile Architecture Revolution
by Jason Bloomberg

www.progress.com

PROGRESS

CHAPTER 1

Introducing Agile Architecture

Agile. Architecture. Revolution. Them's fightin' words, all three of 'em.

Agile—controversial software methodology? Management consulting doublespeak? Word found in every corporate vision statement, where it sits collecting dust, like your grandmother's Hummel figurines?

Architecture—excuse to spend too much on complicated, spaghetti-code middleware? Generating abstruse paperwork instead of writing code that actually works? How to do less work while allegedly being more valuable? (Thanks to *Dilbert* for that last one.)

Revolution—the difference between today's empty marketing drivel and yesterday's empty marketing drivel? Key word in two Beatles song titles, one a classic, the other a meaningless waste of vinyl? What the victors always call wresting control from the vanquished?

No, no, and no—although we appreciate the sentiment. If you're hoping this book is full of trite clichés, you've come to the wrong place. We're iconoclasts through and through. No cliché, no dogma, no commonly held belief is too sacred for us to skewer, barbecue, and relish with a nice Chianti.

We'll deconstruct Agile, and rebuild the concept in the context of real organizations and their strategic goals. We'll discard architectural dogma, and paint a detailed picture of how we believe architecture should be done. And we'll make the case that the Agile Architecture Revolution is a *true* revolution.

Many readers may not understand the message of this book. That's what happens in revolutions—old belief systems are held up to the light so that people can see through them. Some do, but many people do not. For those readers who take this book to heart, however, we hope to open your eyes to a new way of thinking about technology, about business, and about change itself.

Deconstructing Agile

Every specialization has its own jargon, and IT is no different—but many times it seems that techies love to co-opt regular English words and give them new meanings. Not only does this practice lead to confusion in conversations with non-techies, but even the techies often lose sight of the difference between their geek-context definition and the real world definition that "normal" people use.

For example, ZapThink spends far too long defining *Service*. This word has far too many meanings, even in the world of IT—and most of them have little to do with what the rest of the world means by the term. Even words like *business* have gone through the techie redefinition process (in techie-speak, *business* means *everything that's not IT*).

It comes as no surprise, therefore, that techies have hijacked the word *Agile*. In common parlance, someone or something is agile if it's flexible and nimble, especially in the face of unexpected forces of change. But in the world of technology, *Agile* (Agile-with-a-capital-A) refers to a specific category of software development methodology. This definition dates to 2001 and the establishment of the Agile Manifesto, a set of general principles for building better software. The Agile Manifesto (from agilemanifesto.org) consists of four core principles:

1. **Individuals and interactions over processes and tools.** Agile emphasizes the role people play in the technology organization over the tools that people use.
2. **Working software over comprehensive documentation.** The focus of an Agile project is to deliver something that actually works; that is, that meets the business requirements. Documentation and other artifacts are simply a means to this end.
3. **Customer collaboration over contract negotiation.** Customers and other business stakeholders are on the same team, rather than adversaries.
4. **Responding to change over following a plan.** Having predefined plans can be useful, but if the requirements or some other aspect of the environment changes, then it's more important to respond to that change than stick obstinately to the plan.

In the intervening decade, however, *Agile* has taken on a life of its own, as Scrum, Extreme Programming, and other Agile methodologies have found their way into the fabric of IT. Such methodologies indubitably have strengths, to be sure—but what we have lost in the fray is a sense of what is particularly agile about Agile. This point is more than simple semantics. What's missing is

the fundamental connection to agility that drove the Manifesto in the first place. Reestablishing this connection, especially in the light of new thinking on business agility, is essential to rethinking how IT meets the ever-changing requirements of the business.

How do techies know what to build? Simple: Ask the stakeholders (the "business") what they want. Make sure to write down all their requirements in the proverbial requirements document. Now build something that does what that document says. After you're done, get your testers to verify that what you've built is what the business wanted.

Or what they used to want.

Or what they said they wanted.

Or perhaps what they *thought* they said they wanted.

And therein lies the rub. The expectation that the business can completely, accurately, and definitively describe what they want in sufficient detail so that the techies can build it precisely to spec is ludicrously unrealistic, even though such a myth is inexplicably persistent in many enterprise IT shops to this day. In fact, the myth of complete, well-defined requirements is at the heart of what we call the "waterfall methodology, illustrated in Figure 1.1.

In reality, it is far more common for requirements to be poorly communicated, poorly understood, or both. Or even if they're properly communicated, they change before the project is complete. Or most aggravating of all, the stakeholder looks at what the techies have built and says, "Yes, that's exactly what I asked for, but now that I see it, I realize I want something different after all."

Of course, such challenges are nothing new; they gave rise to the family of iterative methodologies a generation ago, including the Spiral methodology, IBM's Rational Unified Process, and all of the Agile methodologies. By taking an iterative approach that involves the business in a more proactive way, the reasoning goes, you lower the risk of poorly communicated, poorly understood, or changing business requirements. Figure 1.2 illustrates such a project.

In Figure 1.2 the looped arrows represent iterations, where each iteration reevaluates and reincorporates the original requirements with any further input the business wants to contribute. But even with the most agile of Agile

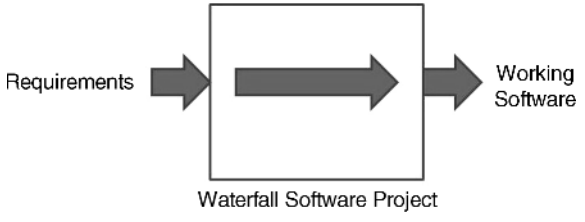


Figure 1.1 Waterfall Software Project

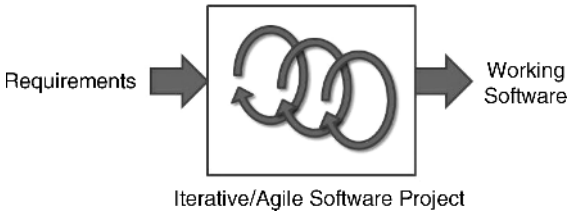


Figure 1.2 Iterative/Agile Software Project

development teams, the process of building software still falls short. It doesn't seem to matter how expert the coders, how precise the stakeholders, or how perfect the development methodology are, the gap between what the business *really* needs and what the software *actually* does is still far wider than it should be. And whereas many business stakeholders have become inured to poorly fitting software, far more are becoming fed up with the entire situation. Enough is enough. How do we get what we *really* want and need from IT?

To answer this question, it's critical to understand that inflexibility is the underlying problem of business today, because basically, if companies (and government organizations) were flexible enough, they could solve all of their other problems, because no problem is beyond the reach of the flexible organization. If only companies were flexible enough, they could adjust their offerings to changes in customer demand, build new products and services quickly and efficiently, and leverage the talent of their people in an optimal manner to maximize productivity. And if only companies were flexible enough, their strategies would always provide the best possible direction for the future. Fundamentally, *flexibility* is the key to every organization's profitability, longevity, and success.

How can businesses aim to survive, even in environments of unpredictable change? The answer is *business agility*. We define business agility as *the ability to respond quickly and efficiently to changes in the business environment and to leverage those changes for competitive advantage*. The most important aspect of this definition is the fact that it comes in two parts: the reactive, tactical part, and the proactive, strategic part. The ability to respond to change is the reactive, tactical aspect of business agility. Clearly, the faster and more efficiently companies can respond to changes, the more agile they are. Achieving rapid, efficient response is akin to driving costs out of the business: It's always a good thing, but has diminishing returns over time as responses get about as fast and efficient as possible. Needless to say, the competition is also trying to improve their responses to changes in the market, so it's only a matter of time til they catch up with you (or you catch up with them, as the case may be).

The second, proactive half of the business agility equation—leveraging change for competitive advantage—is by far the most interesting and powerful part of the story. Companies that not only respond to changes but actually see them as a way to improve their business often move ahead of the competition as they leverage change for strategic advantage. And strategic advantages—those that distinguish one company's value proposition from another's—can be far more durable than tactical advantages.

Building a system that exhibits business agility, therefore, means building a system that supports changing requirements over time—a tall order. Even the most Agile development teams still struggle with the problem of changing requirements. If requirements evolve somewhat during the course of a project, then a well-oiled Agile team can generally go with the flow and adjust their deliverables accordingly, but one way or the other, all successful software projects come to an end. And once the techies have deployed the software, they're done.

Have a new requirement? Fund a separate project. We'll start over and include your new requirements in the next version of the project we already finished, unless it makes more sense to build something completely new. Sometimes techies can tweak existing capabilities to meet new requirements quickly and simply, but more often than not, rolling out new versions of existing software is a laborious, time-consuming, and risky process. If the software is *commercial off the shelf* (COTS), the problem is even worse, because the vendor must base new updates on requirements from many existing customers, as well as their guesses about what new customers will want in the future. Figure 1.3 illustrates this problem, where the software project represented by the box can be as Agile as can be, and yet the business still doesn't get the agility it craves. It seems that Agile may not be so agile after all.

The solution to this problem is for the business to specify its requirements in a fundamentally different way. Instead of thinking about what it wants the software to do, the business should specify how agile it expects the software to be. In other words, don't ask for software that does A, B, C, or whatever. Instead, tell your techies to *build you something agile*.

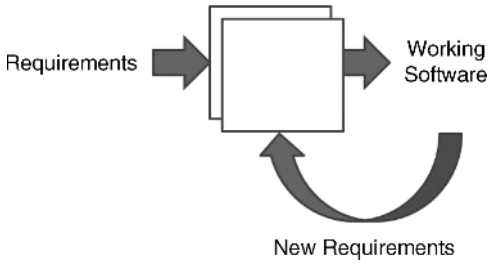


Figure 1.3 Not-so-agile Updates to Existing Software

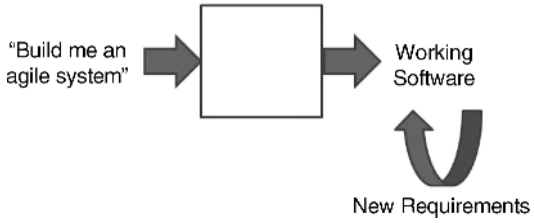


Figure 1.4 What Agile Software Should Really Look Like

We call this requirement the meta-requirement of agility—a *meta-requirement* because agility applies to other requirements: "Build me something that responds to changing requirements" instead of "Build me something that does A, B, and C." If we can build software that satisfies this meta-requirement, then our diagram looks quite different (see Figure 1.4).

Because the software in Figure 1.4 is truly agile, it is possible to meet new requirements without having to change the software. Whether the process inside the box is Agile is beside the point. Yes, perhaps taking an Agile approach is a good idea, but it doesn't guarantee the resulting software is agile.

Sounds promising, to be sure, but the devil is in the details. After all, if it were easy to build software that responded to changing requirements, then everybody would be doing it. But there's a catch. Even if we built software that could *potentially* meet changing requirements, that doesn't mean that it actually *would*—because meeting changing requirements is part of how you would *use* the software, rather than part of how you *build* it. In other words, the *users* of the software must actually be part of the agile system. The box in Figure 1.4 doesn't just represent software anymore. It represents a system consisting of software *and people*.

Architecting Software/Human Systems

Such software/people systems of systems are a core theme of this book. After all, the enterprise—in fact, *any* business that uses technology—is a software/human system. To understand Agile Architecture, it's essential to understand how to architect such a system.

Software/human systems have been with us as long as we've had technology, of course. A simple example is a traffic jam. Let's say you're on the freeway, and an accident in the opposing direction causes your side to slow down. Not because you want to, of course. You're saying to yourself that you really don't care to rubberneck. You'd rather keep moving along. But you can't, because the

person ahead of you slows down. And they're slowing down because the person ahead of them is.

What's going on here? Each vehicle on the freeway is a combination human/technology system: driver and car. The drivers are humans, each making their own choices about how to behave, within the confines of the rules of the road. The traffic jam itself is also a system—what we call a *system of systems*. The traffic jam appears to behave as though it has a mind of its own, independent of the individual decisions of each driver.

Subtract the people from this system and you don't have a traffic jam at all. You have a *parking lot*. And parking lots behave very differently from traffic jams (although sometimes it seems they're one and the same!). Similarly, change the technology, and you have a very different system. Say instead of cars you have trains in a train yard. Train yards might experience traffic jams as well, but they behave very differently from the traffic jams on freeways.

We're not interested in traffic jams in this book, of course. We're interested in *enterprises*. Enterprises are systems of systems as well. We can change the behavior of an enterprise by changing the technology, or by changing human behavior—or some combination of both. Our challenge, then, is *architecting the enterprise* in order to achieve business agility. That's what we mean by Agile Architecture.

The Agile Architecture approach we most frequently talk about is *Service-Oriented Architecture* (SOA). With SOA, IT publishes business Services that represent IT capabilities and information, and the business drives the consumption and composition of those Services. In mature SOA deployments, *policies* drive the behavior of Services and their compositions. If you want to change the behavior, change the policy. In other words, SOA is governance-driven, and governance applies to the behavior of both people and technology.

Agile architectural approaches like SOA, therefore, focus on implementing governance-driven technology/people systems that support changing requirements over time. The challenge, of course, is actually *building* such systems that meet the business agility meta-requirement. Where in this system do we put the agility? It's not in any part of the system. Instead, it's a property of the system as a whole—what we call an *emergent property*. Business agility is an emergent property of the combination technology/human system we call the enterprise.

An emergent property is simply a property of a system as a whole that's not a property of any part of that system. Just as the behavior of a traffic jam consists of properties of the traffic, not of the individual cars, business agility is a property of the enterprise, but not of any of its component systems. We don't look to individual people or technology systems for business agility. We want the organization itself to be agile.

In other words, we started by deconstructing the notion of Agile and ended up with Enterprise Architecture (EA), because what is Enterprise Architecture

but best practices for designing and building the enterprise to better meet changing requirements over time? This is not the static, framework-centric EA from years past that presupposes a final, ideal state for the enterprise. We're talking about a new way of thinking about what it means to architect technology-rich organizations to be inherently agile.

Meta Thinking and Agile Architecture

ZapThink has long bemoaned the Agile Manifesto paradox: that the point to the Manifesto was to be less dogmatic about software development, but today people are overly dogmatic about Agile, defeating its entire purpose. In fact, this paradox has found its way into what is perhaps the most popular of the Agile methodologies: Scrum. Not to worry, all you Scrum aficionados out there; we're not going to teach you how to do Scrum. Instead, we hope to help you think about a broad set of problems in a particular way, starting with *Scrum Buts*.

The notion of a Scrum But arose when it became clear that thousands of organizations were attempting to follow Scrum for their software development projects, but many of them were having problems with one or another of its tenets. As a result, they would say things like:

> "We use Scrum, but Retrospectives are a waste of time, so we don't do them."
>
> or:
>
> "We use Scrum, but we can't build a piece of functionality in a month, so our Sprints are six weeks long."

Retrospectives and Sprints are well-known Scrum best practices (examples from scrum.org). Note that both of these statements follow the same "We use Scrum, but X so Y" pattern, hence the term Scrum But.

The question with Scrum Buts, of course, is what to do with them—or more specifically, how to think about them. There are two schools of thought:

1. Scrum Buts are simply excuses not to do Scrum properly, and if you're not doing it properly, then you're not really doing it at all.
2. Resolving Scrum Buts when they come up in order to achieve the desired result (software that meets the stakeholders' needs) is actually a part of the Scrum methodology. As a result, Scrum Buts are expected and even welcomed.

From our perspective, option #1 is an example of taking a dogmatic approach, which is inherently non-Agile. Option #2 basically says that you can modify the rules if necessary (one of the four principles of the Agile Manifesto), even the rules of Scrum itself.

In other words, option #2 is self-referential—which may be more Agile to be sure, but people have problems with self-reference. It brings up uncomfortable visions of the liar's paradox: "Everything I say is a lie," or in more modern terms, the first rule of Fight Club ("You do not talk about Fight Club.") How can we make sense of the world or anything in it if we have to deal with self-reference paradoxes? If a rule of Scrum is "You can change the rules of Scrum," then couldn't anything and everything be Scrum? What use is that?

Fortunately, such problems of self-reference have a straightforward, if subtle solution. Instead of thinking of such statements as referring to themselves, think of them as actually two separate but related statements, where one relates to the other. We call this meta thinking.

In the case of Scrum Buts, we have the Scrum methodology and the Scrum meta-methodology. A meta-methodology is a methodology for creating methodologies. Remember, the Scrum methodology is a methodology for creating software. The Scrum meta-methodology is a methodology for creating or improving the Scrum methodology. So when someone says:

> "When you say, 'We use Scrum, but we can't build a piece of functionality in a month, so our Sprints are six weeks long,' my recommendation is to try three 30-day Sprints before extending the length of the Sprint."

That entire statement is a Scrum meta-methodology statement. Furthermore, without such statements, your methodology wouldn't be Agile. The obvious conclusion is that all Agile methodologies are actually meta-methodologies. Otherwise they wouldn't be Agile!

We're sure one of you wise guys out there is thinking, what about methodologies for creating meta-methodologies? We'd call those meta-meta-methodologies, of course. And what about methodologies for creating those, *ad infinitum*? We call this the *hall of mirrors problem*, because you only need to have two mirrors facing each other to get the infinite tunnel effect.

Simple answer: We don't need a methodology for creating meta-methodologies. Instead, an informal approach will do. In general, we only want to go to the meta-meta step if there's a bona fide reason to do so, as with Model Driven Architecture (MDA), when they talk about meta-meta-models. But even the brainiacs at the OMG (the standards body that shepherds MDA) don't spend much time thinking about meta-meta-meta-models—at least, we hope not!

Another meta that is central to ZapThink's thinking is the *meta-requirement*. In particular, we're talking about the meta-requirement of business agility as a fundamental driver of SOA, and by extension, Agile Architecture in general. When the business asks for an agile system, they are asking for a system that can respond to changing requirements—which is what makes such agility a meta-requirement.

Finally, at the risk of belaboring the point, let's talk about *meta-architecture*: What does it mean to architect an architecture? Yes, we've been spending a lot of our brain cycles on meta-architecture as well. We've been putting our stamp on how best to do SOA for several years now. Our students may be architecting their organizations and their component systems, but ZapThink has been architecting SOA itself. And now that we can stick a fork in that, it's time to work on architecting Cloud architecture, and more broadly, Agile Architecture.

Defining Architecture: Worse Than Herding Cats

Now that we've explained what we mean by *agile*, let's tackle a toughie: *architecture*. The problem with defining architecture is, well, we're leaving it to architects to come up with the definition. ZapThink loves to point out that the collective term for *architect* is an *argument*. As in a flock of seagulls, a pride of lions, or an argument of architects. Put enough architects together in a room, and sure enough, an argument ensues. Furthermore, architects love nothing more than to argue about the definitions of terms—because after all, definitions are simply a matter of convention. Bring up the question as to what *architecture* means—well, you might as well go home. No more work will be done today!

To avoid such an argument of architects, we're going to use a widely accepted, standard definition of architecture: the Institute of Electrical and Electronics Engineers (IEEE) definition. IEEE defines architecture as "*the fundamental organization of a system embodied by its components, their relationships to each other and to the environment, and the principles guiding its design and evolution*." As you might expect, because architects come up with these definitions, there are actually several standard definitions of architecture. But the IEEE's is perhaps the best known. It's concise, and it contains all the elements of what we think of as architecture.

We will make one tweak to the IEEE definition, however: We're going to interpret it in the plural. So for the purpose of the book, architecture is *the fundamental organization of systems embodied by their components, their relationships to each other and to the environment, and the principles guiding their design and evolution*.

Let's take this definition apart and focus on its key elements to make sure we're all on the same page:

- **Organization of systems.** In other words, architecture is something you do with systems. You organize them. Architecture is something you *do*, not something you *buy*.
- **Environment.** If you look at the enterprise as a system, what is the environment of its component systems? The *people*—the *business itself*. Many architects get this point wrong when they think of the systems as consisting of technology, where the people *use* the technology, as though they were separate from the architecture. In fact, the people are *part* of the system.
- **Evolution.** *Change* is built into the definition of architecture. If you think of an architecture as some diagram you can put on your wall, be it your data architecture, Java architecture, security architecture, or what have you, you're missing the big picture. Such a diagram is at best a *static snapshot in time* of your architecture. To accurately represent an architecture, you must include the principles for the evolution of that diagram.

These fundamental elements of the definition of architecture go beyond IT architectures and Enterprise Architectures. Consider where we got the word *architecture* from: the process of designing buildings and other structures. In fact, the word comes from the word *arch*, because the first architects were the people who knew how to design arches. After all, there's a trick to building an arch: You must provide a temporary support for the arch until the keystone is in place. The first architects were the people who knew this trick.

So, what do building architects actually design? Yes, they must design walls, floors, electrical and plumbing systems and the like—but these are all means to an end. What the building architect actually designs is the *space* defined by those elements, because the space is where the people work or live—in other words, how people get value from the component systems.

Just so with the architecture we're considering in this book. Yes, you have to design the applications, middleware, networks, and so on—but those are all simply means to an end. It's how people *use* those components to achieve the goals of the business that is the true focus of the architect.

Why Nobody Is Doing Enterprise Architecture

There are many flavors of architecture—technical architecture, solution architecture, data architecture, Service-Oriented Architecture, the list goes on and on—but the type this book is most concerned with is *Enterprise*

Architecture. The practice of EA has been around for years, but even the most seasoned practitioners of this craft rarely agree on what EA really is. What's the story? It doesn't help matters that many techies have co-opted the term *Enterprise Architecture* to mean some kind of technology-centric architecture or other. Look up *Enterprise Architect* on a job board and chances are four out of five positions that call themselves "Enterprise Architect" are entirely technology focused. In spite of this confusion, if there's one thing Enterprise Architects can agree on, it's that Enterprise Architecture is not about technology, or at least, not *exclusively* about technology. Sure, every enterprise these days has plenty of technology, but there's more to the enterprise than its IT systems.

Unfortunately, there's little else Enterprise Architects agree on. Some of them point to ontologies like the Zachman Framework, in the belief that if we could only define our terms well enough, we'd have an architecture. Others point to methodologies like the Architecture Development Method (ADM) from The Open Group Architecture Framework (TOGAF), figuring that if we follow the general best practice advice in the ADM, then at least we can call ourselves Enterprise Architects.

Hence, an argument of architects. If you're an architect, you probably already disagree with something we've written. See? What did we tell you?

The problem is, neither Zachman nor TOGAF—or any other approach on the market, for that matter—is truly Enterprise Architecture. Why? Because *nobody is doing Enterprise Architecture*.

The truth of this bold statement is quite obvious when you think about it. Where does Enterprise Architecture take place today? In enterprises, of course. That is, *existing* enterprises. And you don't architect things that already exist. Architecture comes *before* you build something!

Can you imagine hiring an architect *after* building a bridge or a building? I can hear that conversation now: "We built this bridge organically over time, and it has serious issues. So please architect it for us now." Sorry: *too late!*

Most forms of technical architecture don't fall into this trap. A solution architect architects a solution before that solution is implemented. A Java architect or a .NET architect does their work before the coders do theirs. You don't build and then design, you design and then build. Even if you take an Agile, iterative approach, none of your iterations has build before design in it.

Enterprise Architecture, on the other hand, always begins with an existing enterprise. And after working with hundreds of existing enterprises around the world, both private and public sector, we can attest to the fact that every single one of them is *completely screwed up*. You may think that your company or government organization has a monopoly on internal politics, empire building, irrational decision making, and incompetence, but we can assure you, you're not alone.

Enter the Enterprise Architect. The role of today's Enterprise Architect is essentially to take the current enterprise and fix it. OK, maybe not the whole thing, but to make some kind of improvement to it. Go from today's sorry state to some future nirvana state where things are better somehow.

If you're able to improve your enterprise, that's wonderful. You're providing real value to your organization. But you're not doing architecture. Architecture isn't about *fixing* things, it's about establishing a best practice approach to *designing* things.

Okay, so if nobody is doing Enterprise Architecture, then who actually architects enterprises, and what are they actually doing?

The answer: *nobody*. Enterprises aren't architected at all. They are *grown*.

Every entrepreneur gets this fundamental point. When entrepreneurs first sit down to hammer out the business plan for a new venture, they would never dare to have the hubris to architect an organization large enough to be considered an enterprise. There are far too many unknowns. Instead, they establish a framework for growth. Plant the seeds. Water them. Do some weeding and fertilizing now and then. With a bit of luck, you'll have a nice, healthy, growing enterprise on your hands a few years down the road. But chances are, it won't look much like that original plan.

Does that mean there are no best practices for growing and nurturing a startup through all the twists and turns as it reaches the heights of enterprise-hood? Absolutely not. But most people don't consider such best practices to fall into the category of architecture.

What's the difference? "Traditional" Enterprise Architecture—that is, take your massively screwed organization and establish a best practice approach for improving it—follows a traditional systems approach: Here's the desired final state, so take certain actions to achieve that final state.

Growing a business, however, implies that there is no specific final state, just as there is no final state for a growing organism. An acorn knows it's supposed to turn into an oak tree, but there's no specific plan for the oak tree it will become. Rather, the DNA in the acorn provides the basic parameters for growth, and the rest is left up to *emergence*.

Such emergence is the defining characteristic of *Complex Systems*: systems with emergent properties of the system as a whole that aren't properties of any part of the system. Just as growth of living organisms requires emergence, so too does the growth of organizations.

Perhaps it makes sense to call the establishment of best practices for emergence *architecture*. After all, if we can architect traditional systems, why can't we architect complex ones? If we have any hope of figuring out how to actually architect enterprises, after all, we'll need to take a Complex Systems approach to Enterprise Architecture.

Complex Systems: At the Heart of Agile Architecture

Complex Systems are poorly named. In fact, many Complex Systems are quite simple. A Complex System is simply a system that exhibits emergent properties. Complex Systems Theory is particularly fascinating because it describes many natural phenomena, from the human mind to the growth of living creatures to the principle of friction. Furthermore, if you assemble a large enough group of people, they become a Complex System as well, which explains simple emergent properties like a stadium of people doing the wave, to more subtle ones like the wisdom of crowds.

It's important to realize that Complex Systems are actually *systems of systems*. The individual elements of a Complex System are themselves systems, which in turn may be Complex Systems in their own right. However, the individual component systems do not exhibit the emergent properties that the larger Complex System will offer.

Although a large enough group of people will constitute a Complex System in its own right, for our purposes we're looking for innovation in Complex Systems that include some software subsystems. The subsystems are not all software, because people must also be a part of the Complex System we're looking to create. In fact, understanding this basic principle is at the center of the Agile Architecture Revolution.

Traditional software innovation focuses predictably on traditional systems, as opposed to Complex Systems. To design a traditional system, start with the requirements for that system and build to those requirements. As a result, the best you can expect from a traditional system is that it does what it's supposed to do.

The emergent properties that Complex Systems exhibit, however, may be unpredictable—at least, before we build the system to see what behavior emerges. The stickiness property of Velcro, for example, is familiar and predictable to us now, but it took a great leap of innovative imagination to look at the little hooks and loops that make up Velcro and see that enough of them put together would give us the stickiness that makes Velcro so useful. The behavior of stock markets, in contrast, is inherently unpredictable, although it does follow certain patterns that give technical analysts something to base their models on. But if technical analysis accurately predicted market behavior, there would be a lot more billionaire technical analysts in this world!

The wrong approach, therefore, is to build to a set of fixed requirements that will tend to eliminate emergent behavior rather than encourage it. This limitation gives start-ups an advantage, because most traditional IT solutions follow traditional systems approaches that limit their flexibility. For example, traditional integration middleware follows a "connecting things" approach that leads to reduced agility over time, whereas SOA (properly done, not the fake

SOA peddled by the middleware vendors) follows a Complex Systems approach that yields emergent properties like business agility and business empowerment.

So, how do you avoid the wrong approach? Don't build anything that does what it's supposed to do. Instead, build something that will lead to surprises. Not every surprise will be useful, to be sure, so you may have to try a few times before you find an emergent property that people will actually appreciate. Also, remember that any system of systems that has component systems that consist solely of technology will most likely be the wrong approach, as the only surprises you're likely to end up with are bad ones: namely, that the system doesn't even do what it's supposed to do.

The key to successful Agile Architecture is to realize that humans are part of the system, not just users of the system. Although people are unpredictable individually, they are always predictable en masse. Your architecture, therefore, must work at two levels. You must think about individual human/technology subsystems as well as the complex human/technology systems that emerge when you have enough of the subsystems in place. Remember, you can influence human behavior at the Complex Systems level by introducing technology at the component system level. To generate emergent properties at the Complex Systems level, then, you must introduce some element of unpredictability at the component level.

A perfect example of this phenomenon is Twitter. At the component level we have users entering up to 140 characters at a time into a large database that is able to display those tweets based on various search criteria, the default being your own tweet history. However, if that description were all there was to Twitter, it would never have become the sensation it did. It was the fact that people could follow other people, and that people realized they could use hash tags to designate keywords and "at" people with the "@" symbol that introduced an element of unpredictability into the behavior of the individual user/Twitter subsystems. Scale that up to millions of users, and you have the emergent properties inherent in Twitter trending and other aspects of the Twitterverse that make Twitter such a fascinating tool.

Other examples of successful Complex Systems innovations include:

- **SOA governance.** Human activity–centric governance processes supported by a metadata-driven infrastructure enable SOA implementations to exhibit the business agility emergent property. We'll discuss governance in more depth in Chapter 3.
- **Viral marketing.** One person uses a viral marketing tool to tell his or her friends about something cool in a way that encourages them to do the same, leading to the emergence of popularity across large populations.

- **Semantics tooling.** Our computers aren't smart enough to understand the meaning of data, so to effectively automate semantic integration requires human/software subsystems, which leads to the emergence of automated semantic context, at least in theory.
- **Crowdsourcing.** Ask a random person to do something or provide information and there's no telling what you'll get. But use a crowdsourcing tool to ask enough people, and you'll get the emergent property of the wisdom of crowds, where the crowd is able to arrive at the correct answer or solution.
- **Anything with the network effect.** One fax machine or Facebook when it had one user are entirely useless. Two fax machines or Facebook with two users are almost entirely useless. Up the number to three, four, five . . . still pretty damn useless. But at some point, there's a critical mass of users that makes the solution valuable, and you get the emergent property that more people want to join all of a sudden, where before they didn't.

Complex Systems are self-adapting. They are always in a state of change. How would we ever expect to architect a Complex System like an enterprise if we didn't take an Agile Architecture approach that worked at the meta level to deal with change? In fact, the lack of a Complex Systems approach to traditional Enterprise Architecture—architectural approaches that presume a final to-be state—is why all such approaches are inherently flawed.

Our previous Scrum But example is an illuminating illustration of what we mean by an Agile Architectural approach. You could look at a list of Scrum Buts and say, this team is doomed to failure. They've taken the good bits to Scrum and stripped those out, and now they're screwed. Alternatively, you could look at the same list and say, with a few bits of advice about how to deal with the Scrum Buts (in other words, the right meta-methodology), this team might be successful after all.

This admittedly oversimplified scenario has two outcomes: Team crashes and burns, or team is successful in spite of their Scrum Buts. In Complex Systems theory, these outcomes are called *attractors*: Given a set of circumstances, the end result will usually follow one of a set of patterns, in spite of the fact that different people are involved, each with their own skills and preferences. The system is subject to perturbations (the Scrum Buts, in our example), as well as constraints (the advice that makes up the meta-methodology), and the various identities of the people.

Without the appropriate advice, the attractor that is most likely to describe the outcome is the failure attractor. Clearly, the more Scrum Buts you have, and the more serious they are, the more likely your project will fail (although failure is still not certain). But with the proper meta-methodology, you can

steer the project toward the success attractor, in spite of all the Scrum Buts and the various people on the team, who may be disgruntled, incompetent, overworked, or whatever.

Note that the success attractor is not a final state in a traditional sense. Rather, it allows for the fact that perturbations, constraints, and identities are always subject to change. Generalize our Scrum meta-methodology example to the level of the enterprise, and you can get a sense of what we mean by Agile Architecture. Can we design the Complex System of the enterprise, a system consisting of human and technology subsystems, to move toward desirable attractors through the introduction of appropriate meta-policies, meta-processes, and meta-methodologies? That's the million-dollar meta-architecture question.

If you're looking to architect a Complex System, then, our core advice is to come up with a simple tool that is simple when you put it in front of individual users, but yields some kind of unpredictable behavior that when scaled up to large numbers of people delivers an emergent property that people will value. Keep in mind, however, that you hope to be surprised by the result. It may or may not be an emergent property that people will end up wanting, and thus may not be the basis for a viable business model. There is an inherent element of experimental innovation involved, which makes product development inherently risky. On the plus side, however, that risk gives you a barrier to entry, especially against large vendors who sell predictability.

The Complex Systems we're describing are inherently collaborative, because you're including people in the system, and the unpredictability you seek results from allowing them to interact in some way. But remember, the converse is not necessarily true: Not all human/technology systems are Complex Systems. Any multiuser application in the enterprise, for example, combines people and software on some level, but if the vendor didn't build the right unpredictability into it, then it won't exhibit emergent properties.

YOUR COMPLIMENTARY CHAPTER

Copies of *The Agile Architecture Revolution* can be purchased at
http://www.progress.com/agilerevolution

www.progress.com

PROGRESS