Evolutionary architecture and emergent design: Investigating architecture and design

Discovering more-maintainable design and architecture

Skill Level: Intermediate

Neal Ford (nford@thoughtworks.com)
Software Architect / Meme Wrangler
ThoughtWorks Inc.

24 Feb 2009

Software architecture and design generate a lot of conversational heat but not much light. To start a new conversation about alternative ways to think about them, this article launches the *Evolutionary architecture and emergent design* series. Evolutionary architecture and emergent design are agile techniques for deferring important decisions until the last responsible moment. In this introductory installment, series author Neal Ford defines architecture and design and then identifies overarching concerns that will arise throughout the series.

Architecture and design in software have resisted firm definitions for a long time because software development as a discipline has not yet fully grasped all their intricacies and implications. But to create reasonable discourse about these topics, you have to start somewhere. This article series concerns evolutionary architecture and emergent design, so it makes sense to start the series with some definitions, considerations, and other ground-setting.

About this series
This series aims to provide a fresh perspective on the often-discussed but elusive concepts of software architecture and design. Through concrete examples, Neal Ford gives you a solid grounding in the agile practices of *evolutionary architecture* and *emergent design*. By deferring important architectural and design decisions until the last responsible moment, you can prevent unnecessary complexity from undermining your software projects.

Defining architecture

Architecture in software is one of the most talked about yet least understood concepts that developers grapple with. At conferences, talks and birds-of-a-feather gatherings about architecture pack the house, but we still have only vague definitions for it. When we discuss architecture, we're really talking about several different but related concerns that generally fall into the broad categories of *application architecture* and *enterprise architecture*.

Application architecture

Application architecture describes the coarse-grained pieces that compose an application. In the Java world, for example, application architecture describes two things: the combination of frameworks used to build a particular application — which I call the *framework-level architecture* — and the more traditional logical separation of concerns, for which I'm sticking with the *application architecture* moniker. Splitting framework architecture as a distinction matters because most practitioners of object-oriented languages have discovered that individual classes don't work well as a reuse mechanism. (When was the last time you downloaded a single class from the Internet to use in a project?) The unit of reuse in modern object-oriented languages is the library or framework. When you start a new project in framework-rich languages like the Java language, one of the first architectural concerns is the application's framework-level architecture. This style of reuse prevails so deeply in the Java world that I've started saying that we should stop referring to Java programming as an object-oriented language and should call it a framework-oriented language. In many ways, the framework-level architecture represents a physical architecture, described by specific building blocks.

The other interesting aspect of application architecture describes how the logical pieces of the application fit together. This is the realm of design patterns and other structural descriptions, and thus tends to be both more abstract and logical rather than physical. For example, you can say that a Web application adheres to the Model-View-Presenter pattern without specifying which framework you use to achieve that logical arrangement. This logical arrangement is the one most likely to adorn the whiteboards of your workspace as you start working on new parts of the application.

Enterprise architecture

Enterprise architecture concerns itself with how the enterprise as a whole (which usually means the applications running inside a large organization) consumes applications. A common useful metaphor for the relationship between enterprise and application architecture likens *enterprise* to city planning and *application* to building architecture. City planners have to think about getting water, electricity, sewage, and other services to allow the city to function. You can't have one building that consumes more than its share of the water supply. Enterprise architecture ponders the same kinds of things for applications: you can't allow one application to consume all of the network's bandwidth, and if infrastructure services crash, (virtual) sewage backs up.

Enterprise architecture has gotten a lot of attention over the last few years because of Service-Oriented Architecture (SOA). SOA is a massive topic in its own right, so future installments of this series will tackle it as a special case. It has its own interesting aspects because it blurs the lines between enterprise and application architecture when it dictates characteristics of application construction.

The preceding paragraphs offer superficial definitions of these important concepts, but they serve as a launching pad for other more interesting, more nuanced definitions of architecture, including a few definitions harvested from others.

Extant definitions

Many smart people have taken a shot at defining software architecture, so I'm going to let them provide some food for thought. In his classic white paper "Who Needs an Architect?" (see Resources), Martin Fowler discusses several definitions. He quotes the first one from a posting on an Extreme Programming mailing list:

> "The RUP, working off the IEEE definition, defines architecture as 'the highest level concept of a system in its environment. The architecture of a software system (at a given point in time) is its organization or structure of significant components interacting through interfaces, those components being composed of successively smaller components and interfaces.'"

This definition fits nicely within the realm of application architecture as I've described it above. While vague, it does capture the essence of architecture's responsibility: the highest-level concept.

Fowler then quotes Ralph Johnson, who disputed the preceding definition in a reply on the mailing list:

> "A better definition would be: 'In most successful software projects, the expert developers working on that project have a shared understanding of the design system design. This shared understanding is called "architecture." This understanding includes how the system is divided into components and how the components interact through interfaces.'"

Johnson makes an excellent point that software development relies on communication more than technology, and that architecture really represents the shared knowledge about the system, not anything specifically about languages, frameworks, and other technological ephemera.

In the aforementioned paper, Fowler himself provides one of my favorite definitions of architecture:

> "Architecture is about the important stuff. Whatever that is."

This is appropriately vague but oh so descriptive at the same time. Many of the arguments about architecture and design revolve around misunderstanding what is important. What's important to business analysts differs from the important stuff for an enterprise architect. This definition nicely encapsulates that you must define your terms *within your environment* before you can try to define other things.

Fowler's definition also highlights another important aspect of defining something as nuanced as architecture. "The important stuff" not only varies among individuals and groups; those differences can in fact be mutually exclusive. For example, virtually all SOAs make a trade-off between flexibility and speed. The old client/server system you're using now is almost certainly faster than the Web-based, portlet-engine, service-based version replacing it. Unless the new application was written by terrible developers, the extra layers providing the flexibility mean that the response time for users goes up, making it slower for them. Perhaps the architect is the one who gets to tell the users, "Oh, by the way, the new SOA thing we're installing will make it much better for us, but your job will now suck more. Sorry." Maybe that's why architects get paid more than developers.

Which leaves my favorite definition for architecture:

> "Stuff that's hard to change later."

This definition fits best into the idea of an evolutionary architecture. One of the core ideas behind evolutionary architecture is to defer decisions as late as you can, which allows you to substitute alternatives that recent experience has shown are superior. Many of the building blocks of this style of architecture appear throughout this article series and motivated the creation of the series.

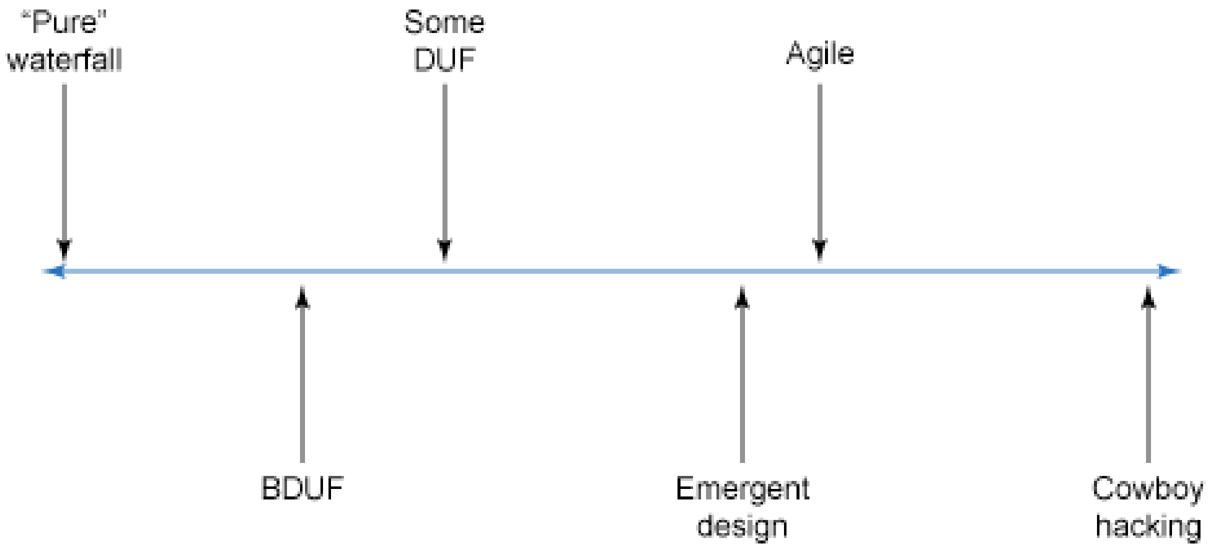
Before leaving the discussion of architecture, I would be remiss if I didn't discuss the "architect" job title. It annoys human resources departments that we as an industry have such poorly defined titles. Many organizations want to promote their best developers — the ones making important decisions about stuff that's hard to change later — but no good industry term exists besides "architect." No common job description exists either, so every company defines what this role means. Some of the architects resemble the Architect at the end of the second Matrix movie (which Fowler categorizes as *Architectus Reloadus*). These architects last wrote code about a decade ago, and now they're making the important decisions for your company. The only software-development tool they use is Visio.

The alternative architect role is one that Fowler calls *Architectus Oryzus* (named after David Rice, one of my colleagues). These architects actively contribute code to the project, pairing with other developers on the hardest parts. Their responsibilities also include interacting with other project stakeholders to make sure that everyone speaks the same language, uses the same definitions, and understands the parts of the system they need to understand. Obviously, this active role is critical to realizing the goals of evolutionary architecture, and thus will appear over and over in this series.

Defining design

Most developers already have a pretty good sense of design, so I won't spend as much time defining it. It represents the nuts and bolts of how a piece of software goes together. It encompasses well-trodden territory such as design patterns, refactoring, frameworks, and other daily developer concerns. Design roughly falls on a spectrum between BDUF (Big Design Up Front) and Cowboy Hacking, as shown in Figure 1:

Figure 1. Spectrum of design



The left side of the spectrum in Figure 1 suggests that you can anticipate all the hundreds and thousands of concerns that pop up when you develop software and tries to limit your responses to them. You'll read much more on this in subsequent installments. Because I'm not spending much time *defining* design doesn't mean that I won't spend a lot of time talking about it. The bulk of this series covers different aspects of how you can allow design to emerge as you develop rather than setting it in stone before you write your first line of code.

Architectural and design concerns

Evolutionary vs. emergent

Notice that this series is called ***Evolutionary** architecture and* ***emergent** design*. Why the distinction between *evolutionary* and *emergent*? Emergent *architecture*, as one of my colleagues pointed out to me, isn't such a hot idea. If you accept the premise that architecture is about things hard to change later, it becomes difficult to allow an architecture to emerge. Architecture concerns infrastructure elements that must exist before you can start the application. However, just because you can't allow architecture to emerge doesn't mean that it can't *evolve*. If you have created a flexible architecture and taken care not to create an irreversible decision, you can allow it to evolve over time as new concerns appear.

With working definitions of architecture and design now at hand, I want to delve into a few overarching areas of concern. All these topics intersect with both architecture and design at a fundamental level, so covering them up front allows me to refer back to them later in the series. First, I discuss technical debt, then complexity, and finally rampant genericness.

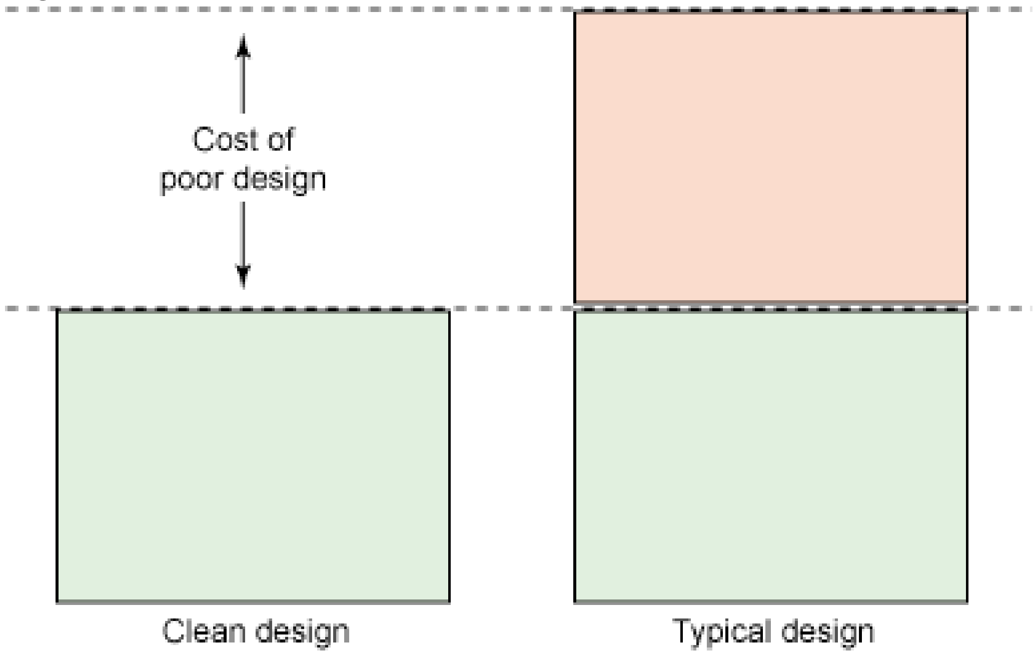
Principal and interest

Every developer becomes aware of the concept of *technical debt*, whereby you make compromises in your design for the sake of some external force, such as schedule pressure. Technical debt resembles credit card debt: you don't have enough funds at the moment, so you borrow against the future. Similarly, your project doesn't have enough time to do something right, so you hack a just-in-time solution and hope to use some future time to come back and retrofit it. Unfortunately, many managers don't seem to understand technical debt, causing resistance to revisiting past work.

Building software isn't like digging a ditch. If you make compromises when you dig a ditch, you just get uneven width or unequal depth. Today's flawed ditch doesn't prevent you from digging a good ditch tomorrow. But the software you build today is the foundation for what you build tomorrow. Compromises made now for the sake of expediency cause *entropy* to build up in your software. In the book *The Pragmatic Programmer*, Andy Hunt and Dave Thomas talk about entropy in software and why it has such a detrimental effect (see Resources). Entropy is a measure of complexity, and if you add complexity now because of a just-in-time solution to a problem, you must pay some price for that for the remaining life of the project.

Let's say that you want to add new features to an existing, long-running project. These new features have a certain inherent complexity to them. However, if you have technical debt already, you must work around those compromised parts of the system to add new features. Thus, the cost for additions mirrors the financial metaphor. Figure 2 shows the difference between the effort required to add a new feature in a cleanly designed system (for example, one with little or no technical debt), versus a typical system that contains a lot of technical debt.

Figure 2. Technical debt and interest



You can think of the inherent complexity as the principal, and the extra effort imposed by previous expedient shortcuts as the interest. Complexity is an interesting subject all by itself.

Essential vs. accidental complexity

Problems that we solve in software have an inherent complexity, which I call *essential* complexity. Complexity arising from the compromises we make that incur technical debt is different. It consists of all the externally imposed ways that software becomes complex, and it shouldn't exist in a perfect world. I call this *accidental* complexity. I define and discuss these terms in depth in my book *The Productive Programmer* (see Resources). These terms generally aren't purely cut and dried: they exist on a spectrum, like design. Some examples will help clarify the distinction.

One of my colleagues worked on a payroll system for a unionized company. One of the concessions that the union secured for some of its members was an extra day off for the start of hunting season. (Hey, they must have had good negotiators.) The employees in question worked at only one factory, but accommodating the extra day off was a legitimate part of the payroll system for the whole company. The change added a lot of complexity to the software, but it was essential complexity because it was part of the business problem to solve.

Another example a little further along the spectrum pops up all the time: field-level security on entry forms. Lots of business people *think* they want fine-grained control of each field's security characteristics. In reality, they almost always hate it when it is implemented because it creates such a burden on the users who need to define and maintain all that metadata. The business people on one of our projects really wanted this feature, so we implemented part of it on one of the screens for them. Once they saw first-hand how much effort was required to make it work, they decided that, because the only access to the application was from a locked office, they could go with more coarse-grained security. This is a great example of a design decision that emerged once the business saw the reality of what they thought they wanted.

At the far end of the spectrum toward accidental complexity is pure plumbing exercises like the first two versions of Enterprise JavaBeans (EJB) technology and tools like BizTalk. A few projects need the extra overhead introduced by these tools, but they do nothing but add complexity to most of the projects that use them.

Three things tend to spawn accidental complexity. I've already discussed the first: just-in-time hacks to code because of schedule or other external pressures. The second is *duplication*, what the Pragmatic Programmers call violations of the DRY (Don't Repeat Yourself) principle. Duplication is the single most insidious diminishing force in software development because it manages to creep into so many places without developers even realizing it. The obvious example is copy-and-paste code, but more sophisticated examples abound. For example, just about every project that uses an object-relational mapping tool (such as Hibernate or iBatis) has lots of duplication. Your database schema, the XML mapping file, and the backing POJOs have slightly different but overlapping information. It is possible to fix this by creating a canonical source for that information and generating the other parts. Duplication harms projects because it resists attempts to make structural changes or refactor toward better code. If you know that you need to change something in three different places, you avoid doing it even if it would make the code better in the long run.

The third enabler of accidental complexity is *irreversibility*. Any decision you make that cannot be reversed will eventually lead to some level of accidental complexity. Irreversibility affects both architecture and design, although its effects are both more common and more damaging at the architectural level. Try to avoid decisions impossible or cumbersome to reverse. One of the mantras I've heard some of my colleagues use is to wait until the *last responsible moment*. This doesn't mean that you should put off decisions too long, but just long enough. What is the last responsible moment you can make a decision about some architectural concern? The longer you can avoid the decision, the more possibilities you leave open for yourself. Ask yourself: "Do I need to make that decision now?" and "What can I do to allow me to defer that decision?" You'll be surprised at the things you can defer until later if you just apply some ingenuity to your decision-making process.

The distinction I made earlier between framework-level architecture and application architecture ties into the principle of Last Responsible Moment. The application architecture tends to be a logical architecture. For example, suppose you know that you want the separation of concerns of Model-View-Presenter. Too often, you make the leap to a physical implementation of that logical architecture by choosing a framework that meets some or all of the requirements. See if you can defer that decision because once you have the physical implementation in place, it constrains the other kinds of decisions you must consider. Putting the framework decision off as long as you can leaves you open to better options that are less polluted by reality.

Rampant genericness

The last of the overarching concerns for architecture and design is a phrase I've made up called *rampant genericness*. We seem to have a disease in the Java world: overengineering solutions by trying to make them as generic as possible. The motivation for this is clear: If we build in lots of layers for extension, we can more easily build more onto it later. However, this is a dangerous trap. Because genericness adds entropy, you damage your ability to evolve the design in interesting ways early in the project. Adding too much flexibility makes every change to the code base more complex.

Of course, you can't ignore extensibility. The agile movement has a great phrase that sums up the decision process for implementing features: YAGNI (You Ain't Gonna Need It). This is the mantra to try to avoid overengineering simple features. Just implement exactly what you need now, and if you need more stuff later, you can add it then. I've seen some Java projects so bloated with compromises in both architecture and design made at the altar of genericness and extensibility that the projects failed. This is of course ironic because planning for the project to live as long as possible truncated its life. Learning how to navigate the fine line between extensibility and overengineering is tough, and it's a subject I'll come back to frequently.

Roadmap

This article contains a lot of hand waving and no source code, which makes it unlike all the other upcoming articles in this series. One of the problems inherent in discussing complex subjects like architecture and design is the context-setting that has to occur to make sure that everyone is on the same page. I've set the stage for the rest of the parts of this series, where I'll delve into specific areas relating to evolutionary architecture and emergent design. Each article will dive deep into a particular illustrative aspect of one or both of these concepts, with lots of detail and source code. Next up: I talk about emergent design through test-driven development, which I've renamed *test-driven design*.

Resources

Learn

- "Who Needs an Architect?" (Martin Fowler, *IEEE Software*, September 2003): Read Fowler's classic white paper.
- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008)): Neal Ford's most recent book expands on several topics in this article.
- *The Pragmatic Programmer* (Andy Hunt and Dave Thomas, The Pragmatic Bookshelf, 2001): This book includes a discussion of entropy's effect on software.
- *Essential XP: Emergent Design* (Ron Jeffries): Web discussion of emergent design considerations in the extreme programming world.
- "Emergent Optimization in Test Driven Design" (Michael Feathers): How testing helps avoid premature optimization.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Neal Ford
Neal Ford is a software architect and Meme Wrangler at **Thought**Works, a global IT consultancy. He also designs and develops applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *The Productive Programmer*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his Web site.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the