Evolutionary Architecture and Emergent Design

Investigating architecture and design

Summary

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.

Defining Architecture

- 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 may be called the *framework-level architecture*
  - the more traditional logical separation of concerns
- object-oriented practitioners have discovered that individual classes don't work well as a reuse mechanism, hence separate out framework architecture

Application Architecture & Frameworks

- 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
- Could say Java has become a framework-oriented language
- Framework-level architecture represents a physical architecture, described by specific building blocks

Application Architecture

- How do 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

Enterprise Architecture

- concerned itself with how the enterprise as a whole uses applications
- application architecture likens *enterprise* to city planning and *application* to building architecture
- Enterprise architecture has gotten a lot of attention because of Service-Oriented Architecture (SOA)
- SOA blurs the lines between enterprise and application architecture when it dictates characteristics of application construction

Extant definitions

- Superficial so far, need to be more nuanced
- Fowler reports “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.’”
- Fits within the realm of application architecture described above. While vague, it does capture the essence of architecture's responsibility: the highest-level concept.

Extant definitions

- Johnson says "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.'“
- software development relies on communication more than technology
- architecture really represents the shared knowledge about the system, not technological ephemera

Extant definitions

- Fowler himself provides a good succinct definition of architecture:

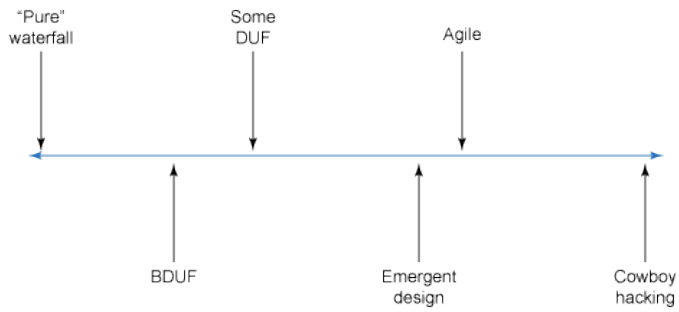
  "Architecture is about the important stuff. Whatever that is."
- What's important to business analysts differs from the important stuff for an enterprise architect
- Or yet another:

  "Stuff that's hard to change later."
  - fits best into the idea of an evolutionary architecture
- A core idea 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

Defining Design

- Represents the nuts and bolts of how a piece of software goes together
- 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
- BDUF 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

Defining Design



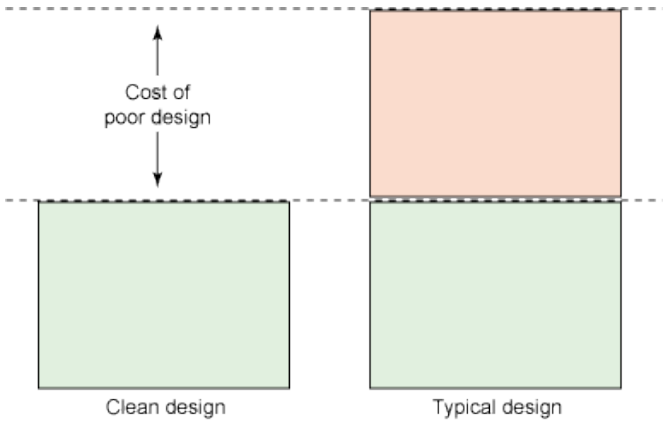
Architectural and design concerns

- Why the distinction between ***evolutionary*** **and** ***emergent***?
- Emergent *architecture* not a good idea. If you accept 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.

Principal and Interest: technical debt

- *technical debt*: make compromises in design for the sake of some external force, such as schedule pressure
- hack a just-in-time solution and hope to use some future time to come back and retrofit it
- compromises made now for the sake of expediency cause *entropy* (measure of complexity) to build up in your software and a price will have to be paid for that for the remaining life of the project
- Can think of inherent complexity as the principal, and the extra effort imposed by previous expedient shortcuts as the interest

Technical debt and interest



Essential versus Accidental Complexity

- problems solved in software have an inherent complexity, which may be called *essential* complexity
- *accidental* complexity arises from the compromises made that incur technical debt
  - externally imposed ways that software becomes complex
- payroll example with extra day off in one factory only adds essential complexity as it is part of business rules
- accidental complexity
  - Just in time hacking
  - pure plumbing exercises like the first two versions of Enterprise JavaBeans (EJB. 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.

Essential versus Accidental Complexity

Three things tend to spawn accidental complexity

- just-in-time hacks
- Duplication is the single most insidious diminishing force in software development
  - can arise from copy-and-paste
  - object-relational mapping tool tend to have lots of duplication. Database schema, the XML mapping file, and the backing POJOs have slightly different but overlapping information.
- The third enabler of accidental complexity is *irreversibility.* Any decision made 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

Principle of Last Responsible Moment

- What can I do to allow me to defer that decision?
- E.g. distinction made earlier between framework-level architecture and application architecture
- suppose you know that you want the separation of concerns of Model-View-Presenter
- too often the leap to a physical implementation of that logical architecture by choosing a framework that meets some or all of the requirements

Rampant Genericness

- overengineering solutions by trying to make them as generic as possible – common in Java world
- build in lots of layers for extension, and later more functionality can be more easily built
- dangerous trap as genericness adds entropy or complexity
- 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

Rampant Genericness

- planning for the project to live as long as possible truncated its life
- YAGNI is an Agile Approach - just implement exactly what you need now, and if you need more stuff later, you can add it then
- how to navigate the fine line between extensibility and overengineering is difficult