Thank you all for being here today. Today we're going to talk about Accidental Architecture, how it happens, how we can seek to correct it, and how to move beyond it in our software projects. To get things started there is a term that we need to define.

And that term is Accidental Architecture. To help explain this term I want to tell a story about a software project.

At the beginning of the project it was bootstrapped. The framework provided enough to get started. And, at the time it seemed like everything had tidy little folders to live in and the team knew where things ought to go.

Then came libraries to help with all all sorts of functionality from authentication, to an admin interface, and countless other little tidbits. At the same time business logic was going in various places based on the conventions and habits encouraged by the framework. Technical debt was being created, but the project was still small enough that it was unclear that it mattered.

Years passed and the project continued on. New functionality was added. Sometimes dead code was removed, and sometimes it was forgotten. Tests were written. What were thought to be reasonable abstractions were created. Mistakes weren't always cleaned up, even when they were recognized.

As the use of the application grew, some parts of the application started to calcify as it became harder to add or remove things in places. The test suite was getting progressively slower. Eventually, no one bothered running the entire test suite locally. And, sometimes event the continuous integration servers weren't entirely trustworthy.

This is not the story of just one project. I've seen this same kind of story played out multiple times. And the fact of every case was that everything seemed OK, until it was clearly not.

When a project is allowed to develop over time without sufficient constraints to keep it maintainable, well organized, and clear in its purpose; it will progress towards such a state of disorganization that it can not be legitimately said that the software demonstrates anything but an accidental architecture. Accidental architecture is where many software projects end up because they are devoid of consistent, disciplined guidance towards anything besides a working state.

Whether or not an Accidental Architecture is acceptable in any given context ultimately represents a value judgement about whether something that works is good enough. There are a lot of factors that contribute to this kind of value judgement and I think its inappropriate to assume that a system that exhibits Accidental Architecture is inherently bad, but it definitely represents a risk for long-term development.

Many of the risks are unknowable until they cause a halt to work. But the notable risks constitute warning signs that the project is in trouble because of Accidental Architecture.

Warning Signs

Changes becoming
Slower,
Further Reaching,
and Less Confident

God Objects

Labelling as “Legacy”

Among the warnings signs to be on the look out for are negative indicators relating to the pace of making changes. [CLICK]

[CLICK] In projects where accidental architecture is taking hold the pace of changes slows down,
[CLICK] Changes also become further reaching,
[CLICK] And, they are made with reduced levels of confidence

All these are related to the sprawling nature of the system’s implementation and the degree of interconnectedness it exhibits.

Another common indicator is the emergence of [CLICK] God Objects. In smaller system it will often be just one, like the User model. But, in large enough systems a pantheon of classes can emerge where logic tends to gather because developers weren’t quite sure where it really belonged.

And, the last indicator I want to address is the tendency to label systems as “Legacy.” Sometimes, there are legitimate reasons for this, but it can be a short-hand for developers disliking working in certain projects so much that they want to label it, and quietly assert that it should be left alone until it can be replaced.

So, once we recognize the nature and signs of accidental architecture, how do we move in another direction?

The key factor is intentionality. Instead of our software architecture being the result of circumstances, it needs to become the result of deliberate choices being made. But, there are risks here too.

Avoiding a Facade

Intentional Effort from
- Individuals
- Teams
- Organization

As we start to turn the corner here and look at how to address accidental architecture it is important to avoid things that will only lead to architectural facades being created. We don't want to create a veneer of intentionality that lacks substance. To do this requires [CLICK] Intentional Effort, which means a unified goal from [CLICK] Individuals, [CLICK] Teams, and [CLICK] the organization as a whole.

One engineer encouraging good practices and sound refactoring can only go so far. The same applies for teams. Without a broad based acceptance of the importance of deliberate architectural thinking there won't be enough institutional inertia to keep things moving in a positive direction.

Bob Martin's *Clean Architecture*

The goal of software architecture is to minimize the human resources required to build and maintain the required system.

The only way to go fast, is to go well.

In the newly released "Clean Architecture" book by Bob Martin he has a couple statements in chapter two that I think can be helpful in shaping the notion of collective interest in deliberate software architecture.

The first is a statement of the goal of software architecture. [CLICK]

And, while this may sound threatening to some it is none the less what I think our goal should be. It also has a happy byproduct in some situations of freeing more resources to work on higher value work, which should be satisfying to all.

The second is a summation of something I think tends to get left out of many discussions around software development, and that is that the developers are stakeholders and should have their voices heard when it comes to how the systems they work are are built. But, that summation emphasizes that development pace and quality are not oppositional traits. [CLICK]

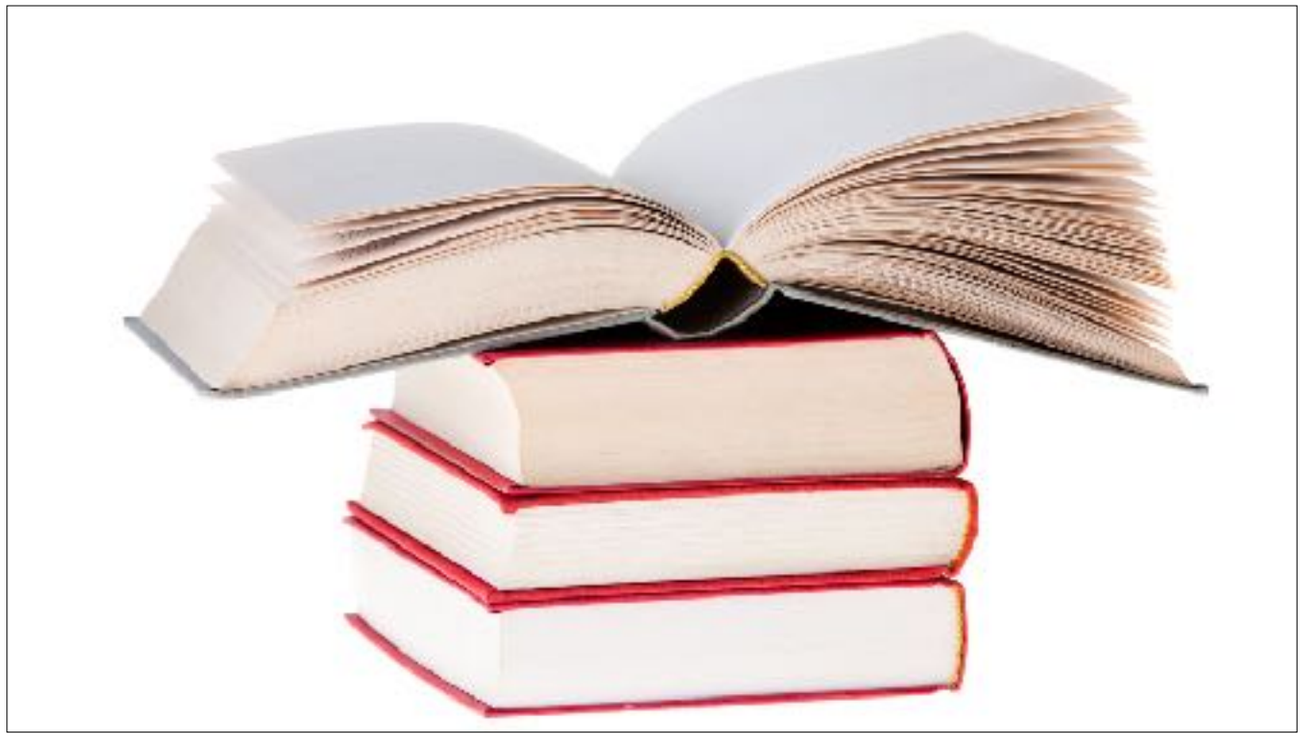
To get to the level of intentionality I think is necessary there has to be a genuine concern for the product being built. This may mean thinking of the software system as a product in the first place. This can be tough for some teams since they may not readily identify with their users, or who they even are.

Eric Evans' "Domain-Driven Design" book is a great resource to emphasize this kind of thinking.

I think this kind of product and customer-oriented thinking is what tends to be difficult for many software developers to grasp and most separates our industry from other engineering disciplines.

Get out to lunch with your team and talk about pain points, and why they exist.

…



Start a book club to start learning about, and discussing different approaches. The number of resources available are staggering and, in my experience, a big part of why teams don’t do things any better is because they don’t know what could be different, or how they could improve things. I’ve got some specific recommendations that will hold until the end.

Form a study group. But, don’t study your project, study software architecture as a broad topic. I went to one of Neal Ford’s hands-on architecture trainings at the O’Reilly Open Source Conference and then translated that material back to my team. We talked about different approaches to software architecture and the applications for our project arose out of the more general nature of the study.

In any given team, or organization, there will be obstacles. The most common are expressions of tribalism or nostalgia. If your team does not already have a strong notion of collective ownership then that will be something to contend with. Ego can be another big challenge, so watch yours. It’s easy when trying to lead others that empathy can wane and communication can become caustic. You can’t control others, so do your best to maintain your own civility, because change is hard.

Another issue is plain old apathy. This can sometimes be connected to nostalgia, but sometimes it’s an issue entirely unto itself. Apathy can be very common from non-technical parties, which is why its important for you to care about the product so that you can be better positioned to understand other points of view and adapt accordingly.

Questions

O'REILLY

Software Architecture

ENGINEERING THE FUTURE OF SOFTWARE

softwarearchitecturecon.com

#OReillySACon

James Thompson

Staff Software Engineer @ Nav

@plainprogrammer

theplainprogrammer.com