Editor: **Robert Blumen**
SalesForce Desk.com
robert@robertblumen.com

Dave Thomas on Innovating Legacy Systems

Sven Johann

A LARGE SHARE of software engineering education (I include Software Engineering Radio among the guilty parties) focuses on new and emerging technologies and methodologies, and most of the thinking around modeling and design is aimed at greenfield projects. Yet in practice, organizations spend most of their resources modifying existing code, some of which is quite old. As an organization grows, the accumulated body of work grows, and the proportion of the organization's accumulated capital in old technology grows along with it. The tradeoffs and constraints facing developers as they work with legacy systems is the theme of episode 242, in which host Sven Johann speaks with ACM Distinguished Member, entrepreneur, and researcher Dave Thomas.

Topics not covered here because of space include project management, scheduling, controlling scoping changes, measuring code quality, outsourcing, and optimizing your investment in test development.

Please contact us with your ideas and thoughts about this column, or any aspect of the show, at team@se-radio.net.
–Robert Blumen

Sven Johann (SJ): What is a legacy system?

Dave Thomas (DT): A legacy system is something that's important to your business because it's generating revenue and keeping your customers happy. It's called a "legacy" for various reasons by young people; they turn up their noses because it's not in their favorite programming language or methodology, perhaps because it's older technology. But it's also a legacy because it's established. And in many cases, it's the core supporting system for the business. It's the legacy of past innovations.

SJ: Why do legacy systems have a bad reputation in software development?

DT: We teach people about greenfield projects. Most new software developers are keen to use the latest technologies and methodologies. We aren't always great at documenting our systems or making them testable. Often, systems that have been around for a long time are very difficult to change. One of the problems, of course, is that any good software probably needs to be killed and rewritten after its third release.

When you find things that haven't been refreshed and redesigned, and they're approaching their seventh or eighth release, they're very difficult to change. Often, they use technologies that people aren't familiar with. They're certainly less malleable and less agile than people would like them to be.

SJ: You mentioned education. Is it necessary to change the curriculum at universities?

0740-7459/16/$33.00 © 2016 IEEE

DT: We teach a lot about programming in the small, which is natural because universities are limited. Having a background in systems engineering or software at the systems level is important because large systems are very different than single applications or websites. It's important to give people awareness of the complexity of reality and that we have to deal with new technologies. That's one of the problems. You need to be able to deal with past technologies because any established business has core systems that were built in previous technologies.

I think it's really more a healthy respect for and an understanding of the strengths and weaknesses of different generations of technologies and methods. It's also important to understand the difference between what a single programmer can do and what large teams of programmers can do. Even the best practices of refactoring are really a joke in the context of a large legacy application. Refactoring tools really don't help you with large legacies.

SJ: How can we deal with legacy systems? I've been in the "big rewrite" multiple times, which mostly failed.

DT: Big rewrites fail. Outsourcing tends to fail. Most of the classic approaches fail. That's because systemic change is very difficult. Outsourcing tends to fail because shipping it to someone else doesn't really change the problem; it might temporarily reduce the economics because the cost of the programmers is cheaper. But the cost of the programmers increases when you send it offshore, because they're getting paid better for better people, and you don't have the domain knowledge.

> Unless you're prepared to develop a substantive body of tests and have the appropriate documentation, it's very difficult to accurately rewrite.

The difference between actually solving a problem that matters to the business by tactically approaching part of the value chain and cracking it is much more important than trying to rewrite an entire application.

SJ: We all think we can just rewrite the whole thing, only better. What are the problems with rewriting?

DT: The rewrite turns out to be a lot more complicated than expected. People typically don't really understand the system before they start changing it. In many systems, you don't have a specification, nor do you have tests. Unless you're prepared to develop a substantive body of tests and have the appropriate documentation, it's very difficult to accurately rewrite.

Refactoring is supposed to be equivalence preserving. But a rewrite is never equivalence preserving because there is always immense pressure during the rewrite to add functionality. In many cases, the rewrite gets you from a system that was in language Y on machine Z to a new system in language C on machine Q that does the same thing. No one wants the same system rewritten in another language. They want a better system.

SJ: You could argue, "It's not the same thing because we reduced technical debt, or we need to have a modern system to retain our good developers." Is that valid?

DT: In my view, neither of those is a valid argument, although certainly they're used. The real issue is that you need a measure to demonstrate that this is true. Only then can you construct a business case. Say you're doing the new code in C++. You might be able to get those developers, but are they really good developers? Are they as good as the ones you had before?

I don't think rewriting an entire system is ever justified. I can see building critical pieces of a new system using new technology, gradually replacing the old system. Those things make sense because they're driven by some clear business value and timeline. A rewrite can take at minimum a year, sometimes two or three years. I don't think any business is interested in waiting for that amount of time. Businesses still like things to happen in a quarter, and you're not going to rewrite much of a major system in a quarter.

SJ: During that time you're also chasing the existing system, because most organizations need to also maintain and enhance the old system.

DT: This often creates a culture of the "tiger team": the people who get

bragging rights because they're programming in the new language with a new technology. Inevitably, they say, "We're gonna do this and this and this." So, the systems and the expectations grow. Managing that is very difficult as well.

The big rewrite is a loser's proposition. It's as stupid as adding more people to a late project. It's much easier to attack those parts of the system where you can deliver a business value or reduce cost, and then apply the innovation from those focus points to gradually change the way your business operates.

SJ: In a previous SE Radio show, we had the example of Twitter. They started writing their application as a Ruby on Rails app. When it stopped scaling, they performed a partial rewrite.

DT: Sometimes people don't want to program in a given language, so they decide that one thing is bad and another thing is good. The problem with Ruby is that the performance and maintainability of the code is more challenging because you can write it a lot faster. Anything you can code [quickly] is both good news and bad news because you have the advantage of getting functioning code quicker, [but you have to maintain more code]. It's a tradeoff. When people use a company like Twitter as an example, they're in the subset of companies that have a lot of money. They can hire the people they want to hire. That presents a different opportunity.

SJ: We discussed some of the bad ways of improving a legacy system. But what are the good ways? You said we have to deliver value. Does that mean we have to understand the business end to end before we can improve anything?

DT: There's a software value chain that is impacted negatively in one way or other. Typically there are critical points in the value chain where making a difference would have a big impact. The approach we favor is that you find the part of the value chain where a bottleneck is or where accelerating or improving it in some way can make a difference. That's where you can employ innovation. If you find a way to change the value chain at that point—in a period of three or four months—then you'll probably get the support of management and you'll be able to deliver.

The approach we use is to [create] a very quick prototype, in a few weeks, to demonstrate that the innovation will actually solve the problem. Then we validate that. We work at scale, because often you can have something that demonstrates an innovation but won't scale. Then we implement it. It's a pretty straightforward approach.

SJ: How do you identify the value chain?

DT: If you're working on something important, ask the senior executives, "What's the most important thing you need changed with regard to your systems?" Usually they know. You can also talk to some of the key developers. Most of the time, you don't have to do a lot of interviews. You should [take] some measurements to validate their assumptions of where the time is spent or where things are slow. People sometimes have intuitions about legacy systems, but it often turns out that the problem is not where they think they it is.

SOFTWARE ENGINEERING RADIO

Visit www.se-radio.net to listen to these and other insightful hour-long podcasts.

Recent Episodes

- 241—Kyle Kingsbury (known as Aphyr on Twitter) talks to host Stefan Tilkov about consensus in distributed systems.
- 243—Host Jeff Meyerson talks with Slava Akhmechet about RethinkDB, an open source database for the real-time Web.
- 245—Author of *Soft Skills* John Sonmez discusses marketing yourself and managing your career with host Charles Anderson.

Upcoming Episodes

- 246—Google's John Wilkes talks with host Charles Anderson about managing large clusters of machines.
- 248—Host Johannes Thönes speaks with Axel Rauschmayer about JavaScript and ECMAScript 6.
- 249—Vaughn Vernon talks to host Stefan Tilkov about reactive programming.

You should be able to find out what the major problem is in two or three weeks. The major issues usually jump out at you. That makes the value clear. The change has to be worth it. In a major organization, you're looking for at least a $10 million problem. I don't like working on things that don't save 20 percent of the total cycle time; it's just not worth it.

Then you have to come up with the right innovation; that's the creative part. At this point, we understand the problem. We understand what the value of improving it would be. If we could improve it by this amount, quickly and fairly inexpensively, then it's worth doing.

The other solution is, "We just rewrite it all (or a portion of it)." And that may be the answer, but it seldom is. That's when you can say, "We see this problem differently." You propose a different way of doing things at this specific point in the value chain that will reduce the cycle time, increase the volume of transactions, or increase the reliability—whatever it is that you're trying to improve.

That's the fun part because that's when you get to innovate. Most legacy systems provide a lot of opportunities for innovation. Typically, and unfortunately, software developers approach [legacy systems] with their current hot technology. Their solution isn't really innovative.

SJ: Goals such as making the code base nicer or easier to maintain are hard to measure. You need measurable goals, right?

DT: Improving the code base so that it's easier to maintain is something I would not give anyone a dollar for. It's so hard to measure and so hard to do. There's always going to be code that people don't like. That's just the way it is, especially when you have a lot of developers. Instead, you're looking for something that's tangible to the business. It's much easier to talk about something that will significantly increase revenue or significantly reduce operating costs. Those are things that you measure. To bring in all-new hammers and saws and new smart people is an innovation of sorts. But it's not innovation that's really unique to that problem. Ask the question, "How could we really change this and get it done very quickly?" You must come up with a clever way of doing it differently.

SVEN JOHANN is a senior consultant at innoQ. Contact him at sven.johann77@gmail.com.

See www.computer.org/software-multimedia for multimedia content related to this article.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscribe to *IEEE Software* by visiting www.computer.org/software.

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.