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

Software Architecture for Developers

Sven Johann

IN EPISODE 228 of Software Engineering Radio, our new host Sven Johann talks with independent consultant Simon Brown, creator of the C4 software architecture model and author of *Software Architecture for Developers*. Brown advocates doing "just enough" design up front and capturing the design through a series of simple, effective diagrams.

Other topics covered in the interview—but omitted from this column because of space—include why the code never matches the diagrams and why frameworks are guilty of this, what you can do to let the code scream the architecture at you, and how to create simple documentation that's easy to maintain and browse. You can download the entire episode at www.se-radio.net.

—*Robert Blumen*

Sven Johann (SJ): You're proposing sketching as an effective way to create and communicate software architecture. Why is communicating software architecture important?

Simon Brown (SB): If you look back 10 or 20 years, we used to have a very process-heavy, document-heavy way of doing design. This is the "go capture all of the requirements, do all of the design, and then document that design in a big, chunky, hefty document" [approach]. Then UML came along and everybody was buying the UML tools and modeling tools, and we were spending months and months just doing design. Fast forward to 12 years ago, and the whole agile thing came along. I think we've gone from one extreme to the other, where a lot of teams are basically doing nothing. What I'm trying to do is push the needle somewhere back toward the middle.

SJ: We had big design up front, then no design up front. Now you're trying to get "just enough" design up front.

SB: Of course, agile doesn't say "don't do design." That's many people's interpretation of the Agile Manifesto and the agile approach, but that's not actually the case. I'm trying to get people thinking about design again and to provide a nice lightweight mechanism that people can use to share things like this. After all, you have this fantastic idea for how to build a software system, and the team needs to understand it. That's the whole purpose of sketching.

Software developers are the biggest stakeholders of software architecture, so that's the primary audience for these sketches. It's really about being able to communicate and share the vision we create of the design or architecture of the thing we want to build. Sketching implies a certain degree of [being lightweight]. What I'm trying to avoid is going into this big model-driven, up-front design process where we have to think through a lot of things. This is about getting the majority of ideas done quickly in a way that's accessible to developers.

SJ: So it's okay to leave stuff out? It's better to be incomplete but lightweight?

0740-7459/15/$31.00 © 2015 IEEE

SOFTWARE ENGINEERING RADIO

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

Recent Episodes

- 232—Internet Engineering Task Force chair Mark Nottingham joins Stefan Tilkov to look back at HTTP's history and explain the protocol's upcoming revision.
- 231—Joshua Suereth and Matthew Farwell discuss SBT (Simple Build Tool) and their new book *SBT in Action* with host Tobias Kaatz.
- 230—Host Jeff Meyerson interviews Shubhra Kar of StrongLoop about server-side programming in JavaScript and how Node.js is changing the game.

Upcoming Episodes

- Author Barry O'Reilly tells host Johannes Thönes what's lean about *Lean Enterprise: How High Performance Organizations Innovate at Scale*.
- Host Robert Blumen quizzes Fangjin Yang, creator of the Druid analytical database, about online analytical processing and making it real-time.
- Host Josh Long and Andrew Clay Shafer discuss the modern platform as a service.

SB: Right. We're definitely looking for some preciseness here, especially around some of the high levels of abstraction, but what we don't want to do is drop down to class-level details. If people are sketching out class diagrams, maybe we have to question why.

SJ: Sketches aren't formalized, so there are probably many ways I can create a sketch, especially ineffective ones.

SB: Part of the work that I do is I get people to design a solution. I have them draw some pictures without much guidance at all. Probably upward of 90 percent of those sketches are ineffective for a number of reasons. On the notational side of things, people don't tend to use UML. If people are not using UML, they are essentially inventing their own abstractions and notations, which causes a lot of confusion. We find a lot of sketches with mixed abstractions, mixed levels of detail. When people have drawn multiple sketches to show different views of the architecture, it's not clear what the relationship is between those different sketches.

SJ: You mentioned UML—is it dead?

SB: One of the questions I regularly ask my audiences at talks and workshops is how many people use UML, and it's about one in 10 people. I've recently had a bunch of audiences where it's been zero people. I use a very small number of UML diagrams for very specific reasons: class diagrams for showing classes and how they interact, activity diagrams to show Web flows, statecharts for states, and sequence and collaboration diagrams to show interactions between things. But that's pretty much it. I don't use UML for architecture diagrams, and I don't see many people using UML tools anymore.

SJ: Let's move on to the C4 model. What is Ben Shneiderman's mantra?

SB: His mantra is basically a way to deal with a lot of data. Essentially, if you have a huge amount of stuff to deal with, what you want to do first is to get an overview. Then you want to zoom and filter, so you're dropping down into a subset of the dataset. Then, if you want more information you can get that on demand. The C4 model is a way of very simply describing the software system. In order to understand the C4 model, you need to backtrack slightly and say, "How do you represent and think about a software system?"

A software system is made up of a number of containers. These are basically deployable or executable units, something you can host code or data in. Those containers contain components. Because I deal with Java and .NET, my components are made of classes. Once you understand that simple tree structure—system, containers, components, classes—you can then draw a diagram at each level. The four Cs in C4 are context, containers, components, and classes.

SJ: When I hear "container" these days, I always think about Docker. Is your definition of a container the same as Docker's?

SB: The simple answer is possibly. Let's imagine you are building a

system and it's made up of a webserver talking to a database. How would you deploy that in terms of Docker containers? You might have a Docker container for the database and a bunch of Docker containers for your website, so in that sense it kind of matches up. What I mean by a container is something that can host code or data. Essentially it's a separately deployable thing, like a webserver, application server, Windows server, standalone application, browser, or mobile client.

SJ: Then the container diagram contains technology choices?

SB: This is kind of hard to show on a podcast without visuals, but if you take the context diagram, you're just zooming into your system. It really just shows the interaction between your mobile app talking to your website, talking to your database, and whatever it is that your system is made up of. It's only showing logical containers.

SJ: Uncle Bob [Robert C. Martin] said the Web is an implementation detail and not part of the architecture. Why should I then include technology choices?

SB: Because, ultimately, you have to build, deploy, and support this thing at some point. If we're building a Web app, we need to store the data somewhere. We are going to have to choose a database and a website. We're going to have to choose a primary language. There are choices that we must make up front. If we're drawing a containers picture as part of our up-front design exercise, maybe the technology choice has not been decided, so maybe we just list our options. But if we're drawing a containers diagram retrospectively to describe an existing system, there's no reason not to put the [technology] choice in.

The major reason why I like putting [technology] choices on architecture pictures is that it brings [them] back down to Earth again. When you flip through documentation in large organizations and they have all these nice, fluffy, architecture pictures, it's just conceptual blobs kind of floating around, talking to one another. Once you start to put [technology] choices on, from my perspective as a developer, I can start to really see and visualize and understand how that thing works in the real world.

> Sketches are about getting the majority of ideas done quickly in a way that's accessible to developers.

SJ: Let's go to components. What's a component in C4?

SB: I've adopted the simplest meaning of component that I possibly could: a bunch of related stuff with a nice, clean interface. I don't want to get involved in discussions around how people decompose their systems and come up with a list of components. That's entirely up to the design process and the preferences they have. For example, this could be a logging component, which could wrap up log4j or commons logging. We could call that a component because it has a task, it has its own responsibilities, and we can create a nice, clean interface on top.

One of the things I talk about in my book is a simple financial risk system. If we were doing component-level design for the risk system, maybe we'd have a risk calculator component, which does all of the heavy lifting and calculations. Maybe there are some data import components, or a data merge component. Maybe there is an alerting and monitoring component. It's that level of granularity.

SJ: Is it only a drawing, or do you also have text?

SB: If you look at UML, it does have a component diagram. There are a couple of varied notations. One has two boxes sticking out one side of the box, and the other has the cup-and-ball notation for [the required and exposed interface], depending on the interface. I find people struggle with that notation, so I keep [it] very simple. On the components diagram, each component is just a box. There is a component name. There's optionally a description of the technology choice. I draft a list of responsibilities to give a flavor of what that component is doing. There are a number of reasons for writing responsibilities in a diagram, but fundamentally, it allows me to have a quick at-a-glance view of a diagram.

In terms of the relationships, I just draw them as single lines—single arrows—and I normally show dependency using the style relationship. If component A depends on component B, I'll draw a line from A to B, with an arrow pointing toward B and a little annotation saying "depends on," "uses," or something like that.

SJ: Because we have no standard notation, everybody comes up with their own notation, right? It's important to say, this is not just an arrow—it has a particular meaning.

SB: If you look at my arrows on my diagrams, they all pretty much point one way, and they're all annotated on the line. So, hopefully, the direction of the arrow and the annotation match up to explain what that relationship is.

SJ: What do effective sketches look like?

SB: Effective sketches are really simple. Make sure you and the team [have] a common way of thinking. Make sure you agree on a set of abstractions, whether that's my C4 modeling or something else, and that you have a way to describe and think about software systems. Once you've done that, understand how you draw a diagram showing each of those things separately—that's what the C4 model does.

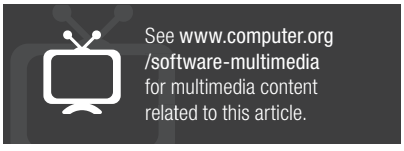
In terms of sketching, it's really simple. Be conscious of notations: if you use different shapes or colors, make sure you have a legend explaining them. Don't leave arrows unannotated; make sure arrows go one way only. Just try to create as simple a solution as possible, and if you do need to highlight different elements, make sure it's explained in the key.

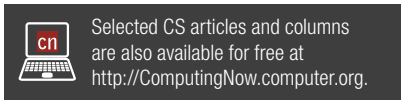
This is basically what maps do. If we get two maps of Amsterdam, they're both showing the same thing. It's the same abstraction, but they use different colors and notations.

SJ: Is there anything important I forgot to ask?

SB: There are a lot of questions around the role of architects. My general approach to architecture is to write code. It probably should come as no surprise, since my website is called Coding the Architecture. For me, it's a very hands-on, collaborative role helping team members, coaching, and mentoring.

SVEN JOHANN is a software developer at Trifork Amsterdam. Contact him at sven.johann@trifork.nl.





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.