

[Home](#)[Users ▾](#)[Submit Article](#)[Voxxed Days](#)[Get Involved](#)[Contact Us](#)[Register](#)[Log In](#)

Search...


VOXXED
DAYS
ZURICH, Switzerland
3 MAR, 2016

[JAVA](#) [MOBILE & WEB](#) [JVM](#) [METHODOLOGY](#) [CLOUD & BIG DATA](#) [FUTURE](#)

Simple Sketches for Diagramming Your Software Architecture

10,848 Views

October 31, 2014

7 Comments

Methodology

simonbrown

Search...

Submit

VOXXED PICKS

Presentation:
Agile
considered
Harmful

METHODOLOGY

PRESENTATIONS

By Simon Brown

If you're working in an agile software development team at the moment, take a look around at your environment. Whether it's physical or virtual, there's likely to be a story wall or Kanban board visualising the work yet to be started, in progress and done. Visualising your software development process is a fantastic way to introduce transparency because anybody can see, at a glance, a high-level snapshot of the current progress.

As an industry, we've become adept at visualising our software development process over the past few years – however, it seems we've forgotten how to visualise the actual software that we're building. I'm not just referring to post-project documentation. This also includes communication during the software development process. Agile approaches talk about moving fast, and this requires good communication, but it's surprising that

[READ PREVIOUS](#)


Ja
Cr
big
de

many teams struggle to effectively communicate the design of their software.

Prescribed methods, process frameworks and formal notations

If you look back a few years, structured processes and formal notations provided a reference point for both the software design process and how to communicate the resulting designs. Examples include the Rational Unified Process (RUP), Structured Systems Analysis And Design Method (SSADM), the Unified Modelling Language (UML) and so on. Although the software development industry has progressed in many ways, we seem to have forgotten some of the good things that these older approaches gave us.

In today's world of agile delivery and lean startups, some software teams have lost the ability to communicate what it is they are building and it's no surprise that these teams often seem to lack technical leadership, direction and consistency. If you want to ensure that everybody is contributing to the same end-goal, you need to be able to effectively communicate the vision of what it is you're building. And if you want agility and the ability to move fast, you need to be able to communicate that vision efficiently too.

Abandoning UML

As an industry, we do have the Unified Modelling Language (UML), which is a formal standardised notation for communicating the design of software systems. I do use UML myself, but I only tend to use it sparingly for sketching out any important low-level design aspects of a software system. I don't find that UML works well for describing the software architecture of a software system. While it's possible to debate this, it's often irrelevant because many teams have already thrown out UML or simply don't know it.

Such teams typically favour informal "boxes and lines" style sketches instead but often these diagrams don't make much sense unless they are accompanied by a detailed narrative, which ultimately slows the team down. Next time somebody presents a software design to you focussed around one or more informal sketches, ask yourself whether they are presenting what's on the sketches or whether they are presenting what's still in their head.

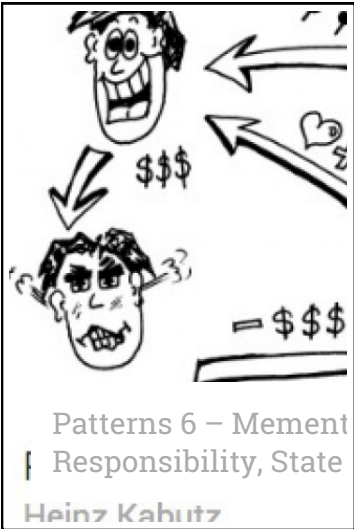


METHODOLOGY

INTERVIEWS



METHODOLOGY COURSES

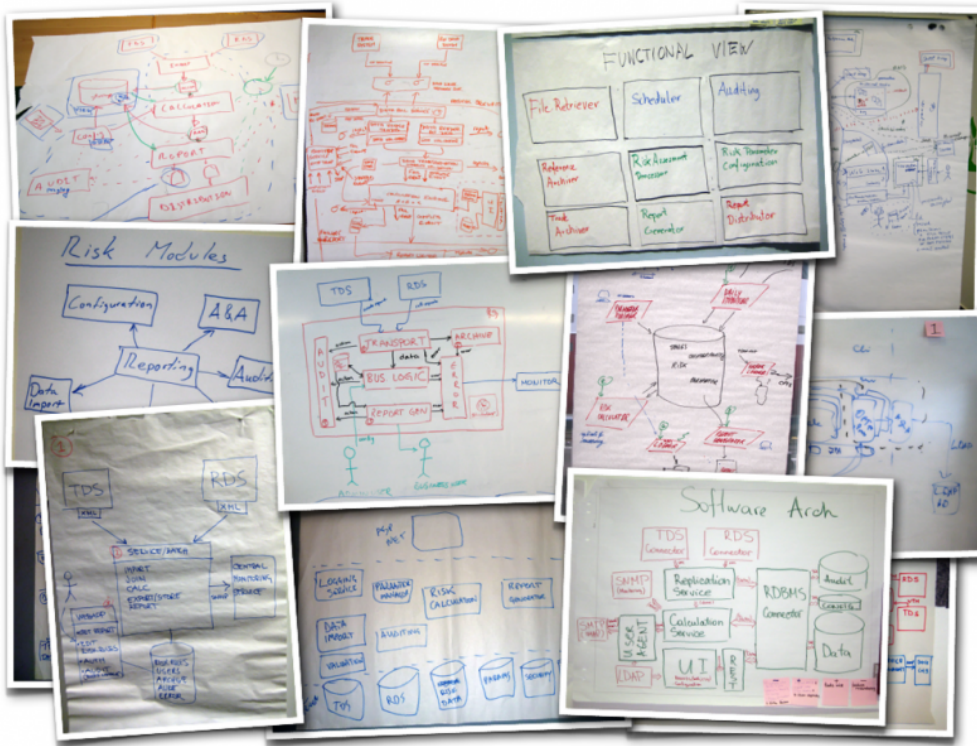


MOST POPULAR

LAST 24HRS

LAST 7 DAYS

ALL-TIME

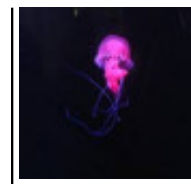


Abandoning UML is all very well but, in the race for agility, many software development teams have lost the ability to communicate visually. The example software architecture sketches (above) illustrate a number of typical approaches to communicating software architecture and they suffer from the following types of problems:

- Colour-coding is not explained or is inconsistent.
- The purpose of diagram elements (i.e. different styles of boxes and lines) is not explained.
- Key relationships between diagram elements are missing or ambiguous.
- Generic terms such as “business logic”, “service” or “data access layer” are used.
- Technology choices (or options) are omitted.
- Levels of abstraction are mixed.
- Diagrams try to show too much detail.
- Diagrams lack context or a logical starting point.

Some simple abstractions and the C4 model

Informal boxes and lines sketches can work very well, but there are many pitfalls associated with communicating software designs in this way. My approach is to use a small collection of simple diagrams that each show a different part of the same overall story. In order to do this though, you need



**Reasons
the
Future of
Tcl is
Bright**

132 views



**Raspberry Pi Rivals:
New arrivals in 2015
(Part Two)**

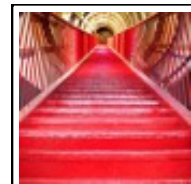
84 views



Do Good

**Microservices
Architectures Spell the
Death of the Enterprise
Service Bus?**

72 views



**Scala
Tutorial:
Getting
Started
with**

Scala

35 views



**Code
Java on
the**

Raspberry Pi (Part One)

35 views

RECENT POSTS

to agree on a simple way to think about the software system that you're building.

Assuming an object oriented programming language, the way that I like to think about a software system is as follows: a software system is made up of a number of containers, which themselves are made up of a number of components, which in turn are implemented by one or more classes. It's a simple hierarchy of logical technical building blocks that can be used to illustrate the static structure of most of the software systems I've ever encountered. With this set of abstractions in mind, you can then draw diagrams at each level in turn. I call this my **C4 model**: context, containers, components and classes. Some diagrams will help to explain this further.

Context diagram

A context diagram can be a useful starting point for diagramming and documenting a software system, allowing you to step back and look at the big picture. Draw a simple block diagram showing your system as a box in the centre, surrounded by its users and the other systems that it interfaces with.

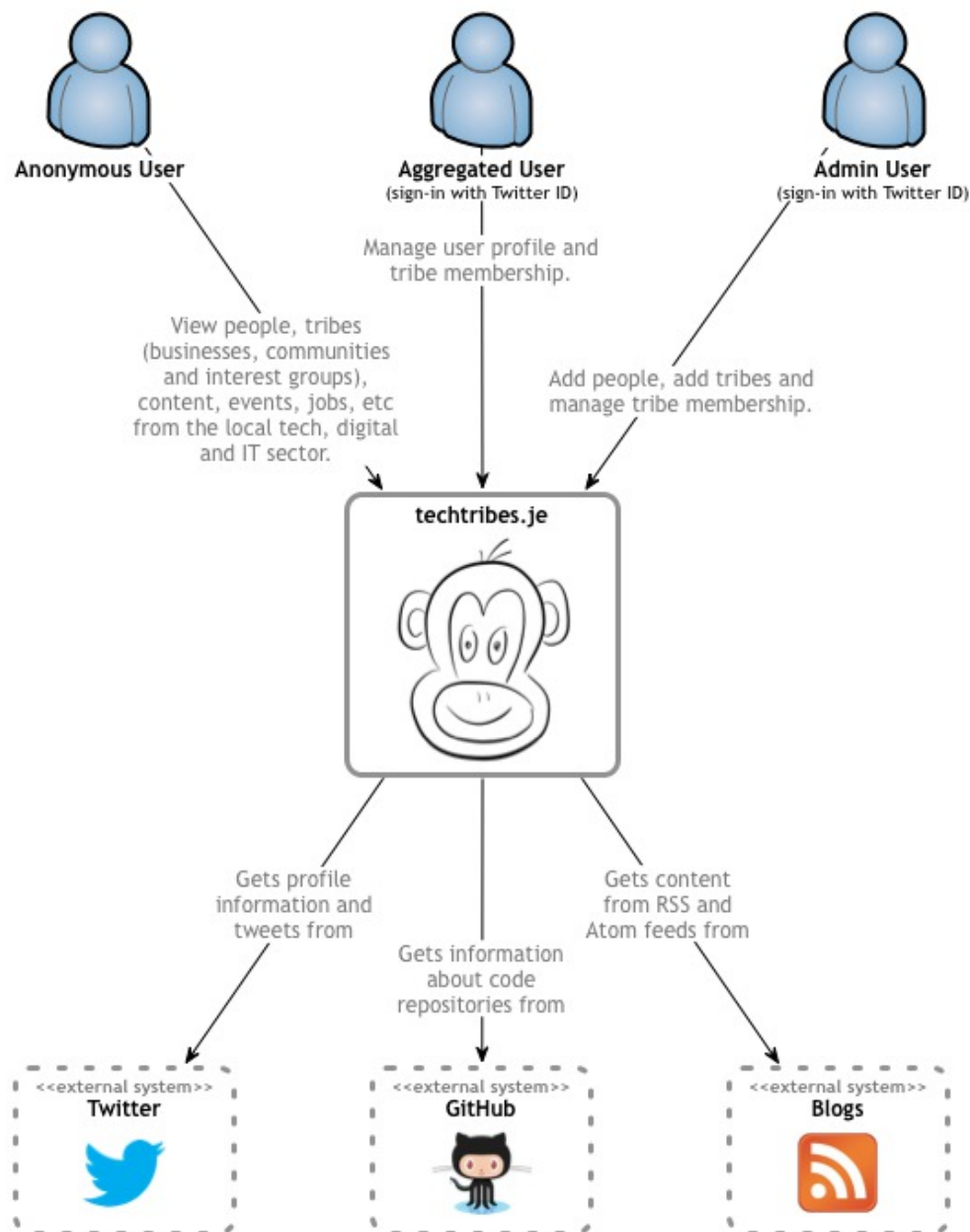
Let's look at an example. The techtribes.je website provides a way to find people, tribes (businesses, communities, interest groups, etc) and content related to the tech, IT and digital sector in Jersey and Guernsey, the two largest of the Channel Islands. At the most basic level, it's a content aggregator for local tweets, news, blog posts, events, talks, jobs and more. Here's a context diagram that provides a visual summary of this.

7 "Must Watch" DevOps Videos

10 Insights from Doing International Recruiting
Walmart Wants to Check You Out of Cloud Hotel California

7 Reasons the Future of Tcl is Bright

JavaScript – A Tough Love



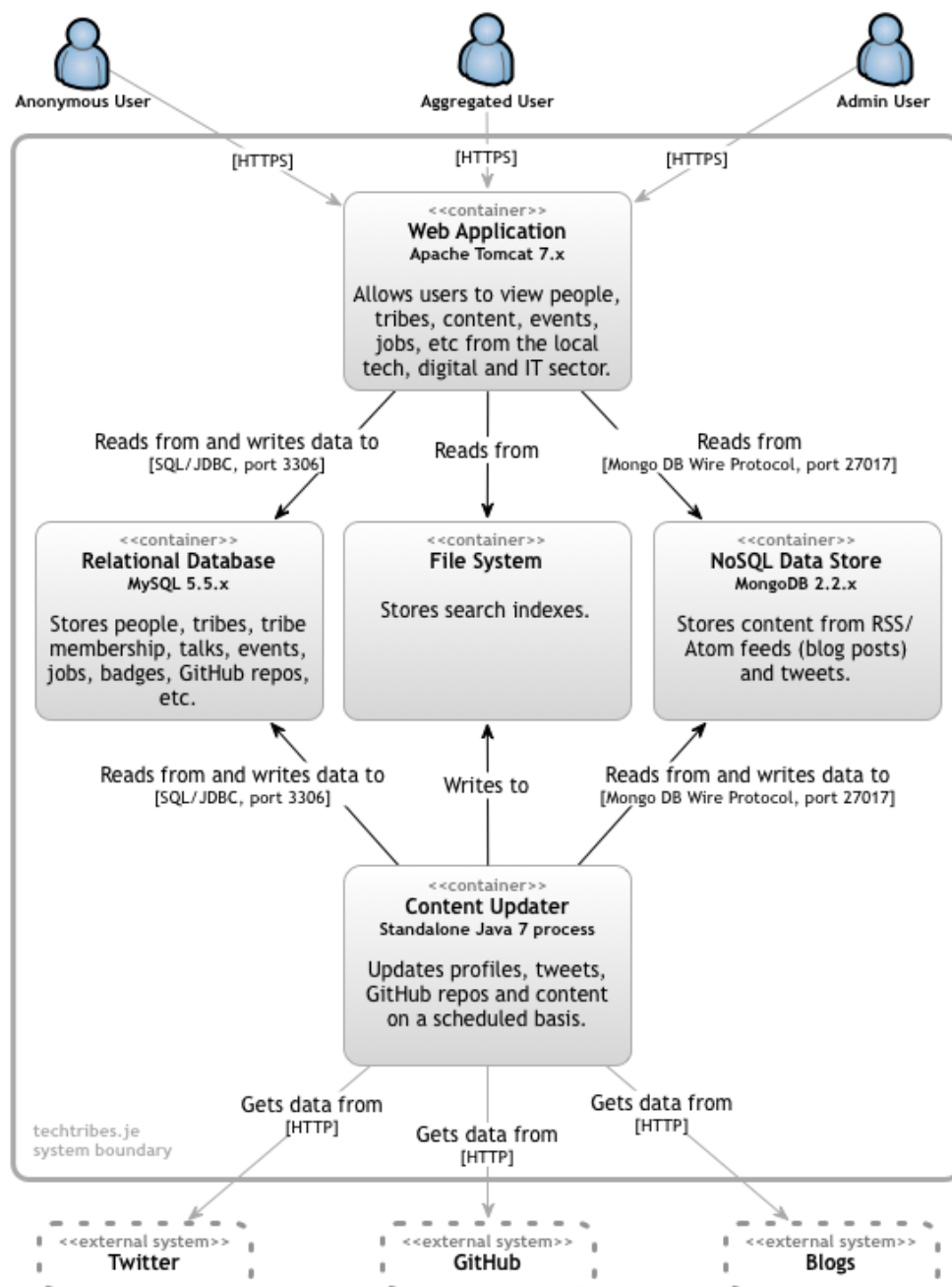
techtribes.je - Context

Detail isn't important here as this is your zoomed-out view showing a big picture of the system landscape. The focus should be on people (actors, roles, personas, etc) and software systems rather than technologies, protocols and other low-level details. It's the sort of diagram that you could show to non-technical people.

Containers diagram

Once you understand how your system fits in to the overall IT environment with a context diagram, a really useful next step can be to illustrate the high-level technology choices with a containers diagram. By "container" I mean something like a web application, mobile application, database, file system,

etc. Essentially, a container is anything that can host code or data; it's an execution environment or data storage. The following diagram shows the logical containers that make up the techtribes.je website.



techtribes.je - Containers

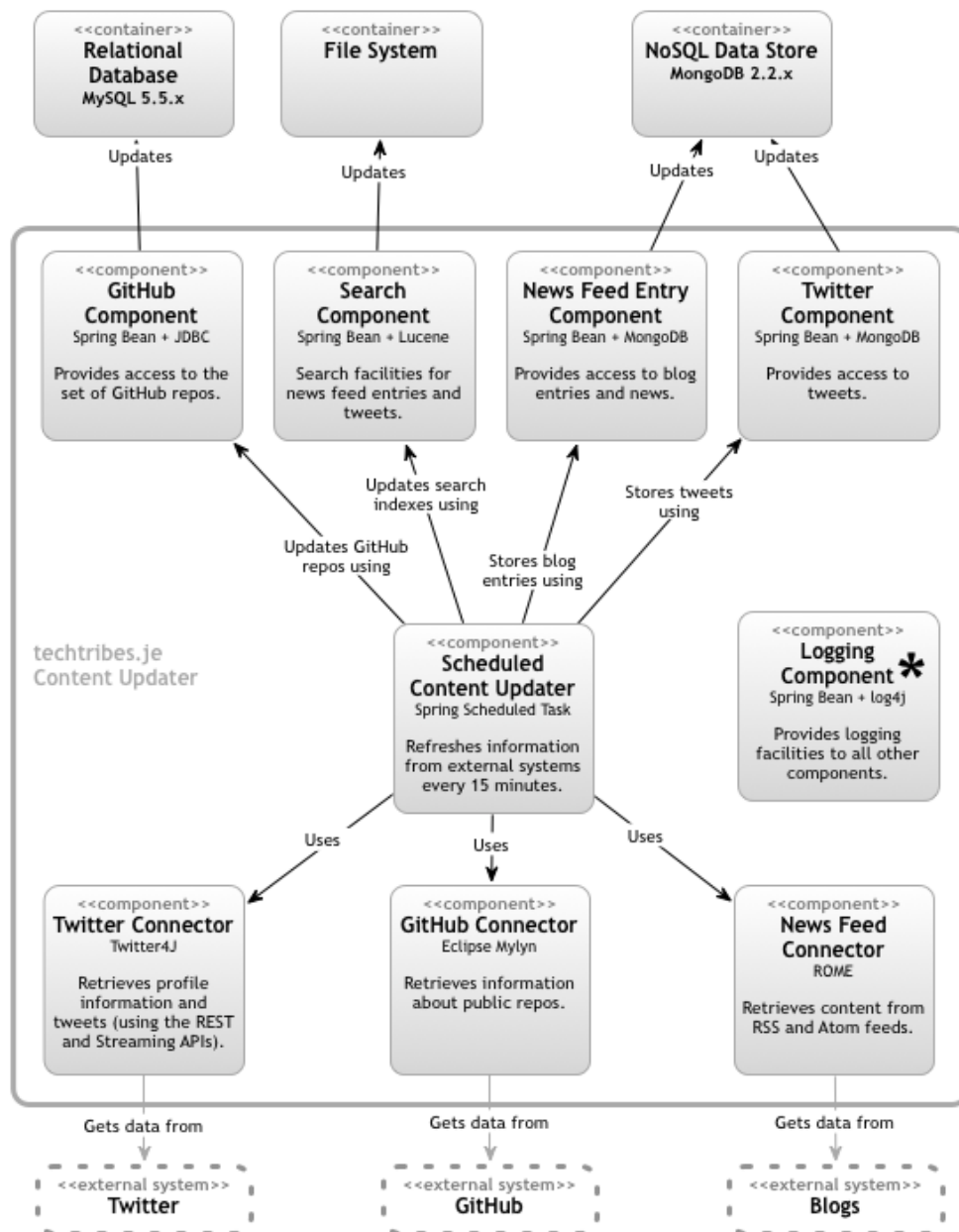
Put simply, techtribes.je is made up of a web application running on Apache Tomcat that provides users with information, with that information being kept up-to-date by a standalone content updater process. All data is stored either in a MySQL database, a MongoDB database or the file system. It's worth pointing out that this diagram says nothing about the number of physical instances of each container.

For example, there could be a farm of web servers running against a MongoDB cluster, but this diagram doesn't show that level of information. Instead, I show physical instances, failover, clustering, etc on a separate deployment diagram. The containers diagram shows the high-level shape of the software architecture and how responsibilities are distributed across it. It also shows the major technology choices and how the containers communicate with one another. It's a simple, high-level technology focussed diagram that is useful for software developers and support/operations staff alike.

Components diagram

Following on from a containers diagram showing the high-level technology decisions, I'll then start to zoom in and decompose each container further. However you decompose your system is up to you, but I tend to identify the major logical components and their interactions. This is about partitioning the functionality implemented by a software system into a number of distinct components, services, modules, subsystems, layers, workflows, etc.

As illustrated by the containers diagram, techtribes.je includes a standalone process that pulls in content from Twitter, GitHub and blogs. The following diagram shows the high-level internal structure of the content updater in terms of components.



techtribes.je - Components - Content Updater
Standalone Java Process

* Used by all components

This diagram shows that the content updater is made up of a number of components. A scheduled content updater component uses a Twitter connector, a GitHub connector and a news feed connector to retrieve information from the outside world. It then also uses some additional components to sort this information into the appropriate data store. This diagram shows how the content updater is divided into components, what each of those components are, their responsibilities and the technology/implementation details.

Class diagrams

This is an optional level of detail and I will typically draw a small number of high-level UML class diagrams if I want to explain how a particular pattern or component will be (or has been) implemented. The factors that prompt me to draw class diagrams for parts of the software system include the complexity of the software plus the size and experience of the team. Any UML diagrams that I do draw tend to be sketches rather than comprehensive models.

Think about the audience

There seems to be a common misconception that “architecture diagrams” must only present a high-level conceptual view of the world, so it’s not surprising that software developers often regard them as pointless. In the same way that software architecture should be about coding, coaching and collaboration rather than ivory towers; software architecture diagrams should be grounded in reality too. Including technology choices (or options) on the diagrams helps prevent diagrams looking like an ivory tower architecture where a bunch of conceptual components magically collaborate to form an end-to-end software system.

A single diagram can quickly become cluttered and confused, but a collection of simple diagrams allows you to easily present the software from a number of different levels of abstraction. And this is an important point because it’s not just the software developers within the team that need information about the software. There are other stakeholders and consumers too; ranging from non-technical domain experts, testers and management through to technical staff in operations and support functions. For example, a diagram showing containers is particularly useful for people like operations and support staff that want some technical information about your software system, but don’t necessarily need to know anything about the inner workings.

Common abstractions over a common notation

The goal with these sketches is to help teams communicate their software designs in an effective and efficient way rather than creating another comprehensive modelling notation. UML provides both a common set of abstractions and a common notation to describe them, but I rarely find teams that are using either effectively. I’d rather see teams able to discuss their software systems with a common set of abstractions in mind rather than struggling to understand what the various notational elements are trying to show.

For me, a common set of abstractions is more important than a common notation. Most maps are a great example of this principle in action. They all tend to show roads, rivers, lakes, forests, towns, churches, etc but they often use different notation in terms of colour-coding, line styles, iconography, etc. The key to understanding them is exactly that, a key/legend tucked away in a corner somewhere. We can do the same with our software architecture diagrams.

It's worth reiterating that because you're creating your own notation rather than using a standard like UML, informal boxes and lines sketches provide flexibility at the expense of diagram consistency. My advice here is to be conscious of colour-coding, line style, shapes, etc and let a set of consistent notations evolve naturally within your team. Including a simple key/legend on each diagram to explain the notation will help. Oh, and if naming really is the hardest thing in software development, try to avoid a diagram that is simply a collection of labelled boxes. Annotating those boxes with responsibilities helps to avoid ambiguity while providing a nice at a glance view.

“Just enough” up front design

As a final point, Grady Booch has a great explanation of the difference between architecture and design. He says that architecture represents the “significant decisions”, where significance is measured by cost of change. The context, containers and components diagrams show what I consider to be the significant structural elements of a software system. Therefore, in addition to helping teams with effective and efficient communication, adopting this approach to diagramming can also help software teams that struggle with either doing too much or too little up-front design. Starting with a blank sheet of paper, many software systems can be designed and illustrated down to high-level components in a number of hours or days rather than weeks or months.

Illustrating the design of your software can be a quick and easy task that, when done well, can really help to introduce technical leadership and instil a sense of a shared technical vision that the whole team can buy into. Sketching should be a skill in every software developer's toolbox. It's a great way to visualise a solution and communicate it quickly plus it paves the way for collaborative design and collective ownership of the code.


Software Architecture for Developers

Technical leadership by coding, coaching, collaboration,
architecture sketching and just enough up front design

Software Architecture for Developers

Duration: 2:40:09

Details



★★★★★ (2 votes, average: 5.00 out of 5)

You need to be a registered member to rate this post.

TAGGED agile agility Analysis And Design Method (SSADM)

architecture C4 model SSADM UML

Unified Modelling Language (UML)

ABOUT AUTHOR



simonbrown

[View all posts](#)

Jet-setting software geek; speaker, author of "Software Architecture for Developers" (<http://t.co/AxD2OWG56r>). Tech leadership and the C4 architecture model.

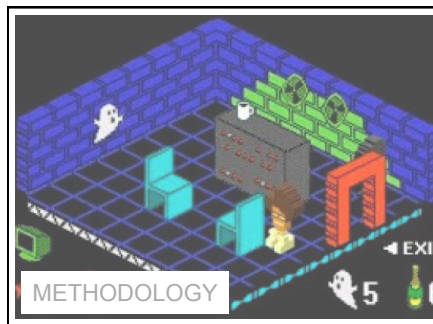
  

RELATED ARTICLES



February 25, 2015

**A Conversation with a Camel
Committer**



December 17, 2014

**Self-Healing Applications: Are
They Real?**



**Und
E**

7 COMMENTS



Ramesh May 20, 2015

Simplifying design diagrams by 4C is nice thought. your diagrams looks good. there are lot of tools out there but I am wondering what tool you are using to create this diagram .. if you recommend some simple tools to make diagram very fast that would be helpful. Thanks

[Reply](#)



simonbrown May 21, 2015

Hi Ramesh,

Thanks! Those diagrams were drawn using OmniGraffle for the Mac, but they are easily do-able in Microsoft Visio or other similar tooling. From my perspective, recommending a drawing tool for creating software architecture diagrams in 2015 is crazy, so I'm actually building some tooling to do this. It's called Structurizr (<https://www.structurizr.com>) and allows you to create simple, versionable software architecture diagrams based upon a model, part of which can be extracted from the code. You can find some background to this approach in this talk -> <http://www.ustream.tv/recorded/61488372>

Cheers

Simon

[Reply](#)



Michael 'Monty' Montgomery July 06, 2015

For me, to get your Dev community really rockin' you need to provide consistency and most of all repeatability in your approaches to both system decomposition (decomp) and runtime modeling.

Have you ever checked out The IDesign Method (www.idesign.net)?

It's both a straightforward methodology for decomposing a service-oriented (SO) problem space and a clear, lightweight modeling notation for capturing key SO design decisions that allows you to express runtime behavior in a way that Devs can actually build against. The last part is priceless.

The approach also defines a terse, tech-agnostic SO taxonomy

that is perfect for building microservice-based systems and allows your Dev community to settle in on a simple pattern-based architectural language they can grok and use to communicate their ideas effectively and efficiently.

This all combines into a tight ‘just enough’ decomp approach and modeling style that allows you, your architecture and your teams to BE agile, instead of just ‘doing’ agile.

I’ve found that providing your Dev community a firm foundation upon which to build does not stifle creativity. Just the opposite. It acts as a springboard to unlock their ideas by getting rid of the drudgery and day-to-day turbulence caused by the impedance mismatch between design and code.

Cheers!

-Mont

[Reply](#)



Martyr2 July 21, 2015

I fail to see how this “C4 model” as you call it is all that much different than your typical data flow diagram. During system design classes in university we started with the context diagram and had level 1, level 2 and level 3 DFDs that broke down systems to components to entities and classes. I can’t say this approach you are presenting is all that different than that.

[Reply](#)



Hendrik September 12, 2015

We’ve just finished an in-house training by Simon @ Nedap (<http://www.nedap.com>). The C4 model was a part of that training. Challenges, as you call them, are not explicitly part of C4, but with did a nice risk assessment exercise: each team member got to stick stickies on the diagrams indicating where he saw the risk (or challenge).

The funny thing about challenges is that they are quite personal. Everybody sees different challenges. This is also reflected during for example planning poker, where team members can estimate quite differently from time to time. I think putting them in the model has the downside that the challenges get “real”, whereas discussing them may lead to changes in the design.

[Reply](#)



hendrik September 12, 2015

My reply above was actually ment to be reply to Charlie Alfred below. Sorry about that.

[Reply](#)



Charlie Alfred August 31, 2015

Hi Simon,

C4 is a nicely explained approach. Well done! Two comments:

1. How does Context handle variability? Simple examples

o stationary users vs. mobile

o Cloud deployment vs. onsite.

The variations in these two pairs of contexts can have a material effect on arcitecture and system capability.

2. Is there room for a 5th C in your model: Challenges?

Challenges explain the obstacles and risks in a problem, by linking business and technical drivers (including unknowns and dependencies) to significance (i.e. importance of product goals and capabilities).

At a minimum, this exposes the WHY's in a system for all to see, think about and discuss. Also, prioritizing challenges can be a great help in determining the story sequence. In my experiences with agile, this activity can be fraught with invisible difficulties

Best,

Charlie

[Reply](#)

LEAVE A REPLY

Connect with:



Leave a Reply

Your email address will not be published. Required fields are marked *

Name:*

E-mail:*

Comment:*

JAVA MOBILE & WEB JVM METHODOLOGY CLOUD & BIG DATA FUTURE



Voxxed is a knowledge sharing platform with the same DNA as the DevOxx conferences and a productive relation with Parleys.com.



4th Floor, 27-33
Bethnal Green
Road, London.
E16LA - UK



+44 (0) 207 613
4688



info@voxxed.com

HOSTED BY CLEVER-
CLOUD

We like to thank Clever-
Cloud for hosting
Voxxed.com

FOLLOW US



RECENT POSTS

7 "Must Watch" DevOps
Videos October 16, 2015
10 Insights from Doing
International Recruiting
October 16, 2015
Walmart Wants to Check
You Out of Cloud Hotel
California October 15,

TAG CLOUD

java ee NoSQL IoT
Java 8 Streams API
SQL linux container
coding Lambdas
Docker JSR devops
Red Hat data code issues
Apache Spark big data

2015

7 Reasons the Future of
Tcl is Bright October 15,
2015

JavaScript – A Tough
Love October 15, 2015

jvm Threading
Java 8 developer best
practices internet of
things microservices
java JDK cloud
software Android Java
SE Parallel
Streams programming
methodology scala
programming
languages JCP
Strings paas
Cassandra google
Apache Foundation
oracle

© 2015 Copyright Voxxed. All Rights reserved.

