Broadening the T

www.codingthearchitecture.com

Simon Brown

Hands-on software architect

C5 alliance

coding {the} architecture

Skills Matter

is for technology

Deep hands-on
technology skills

We understand

programming language syntax

We understand

APIs and frameworks

We understand

design patterns

We understand

automated unit testing

But we typically use a

single

technology on a day-to-day basis

[Microsoft | Java | Open Source]

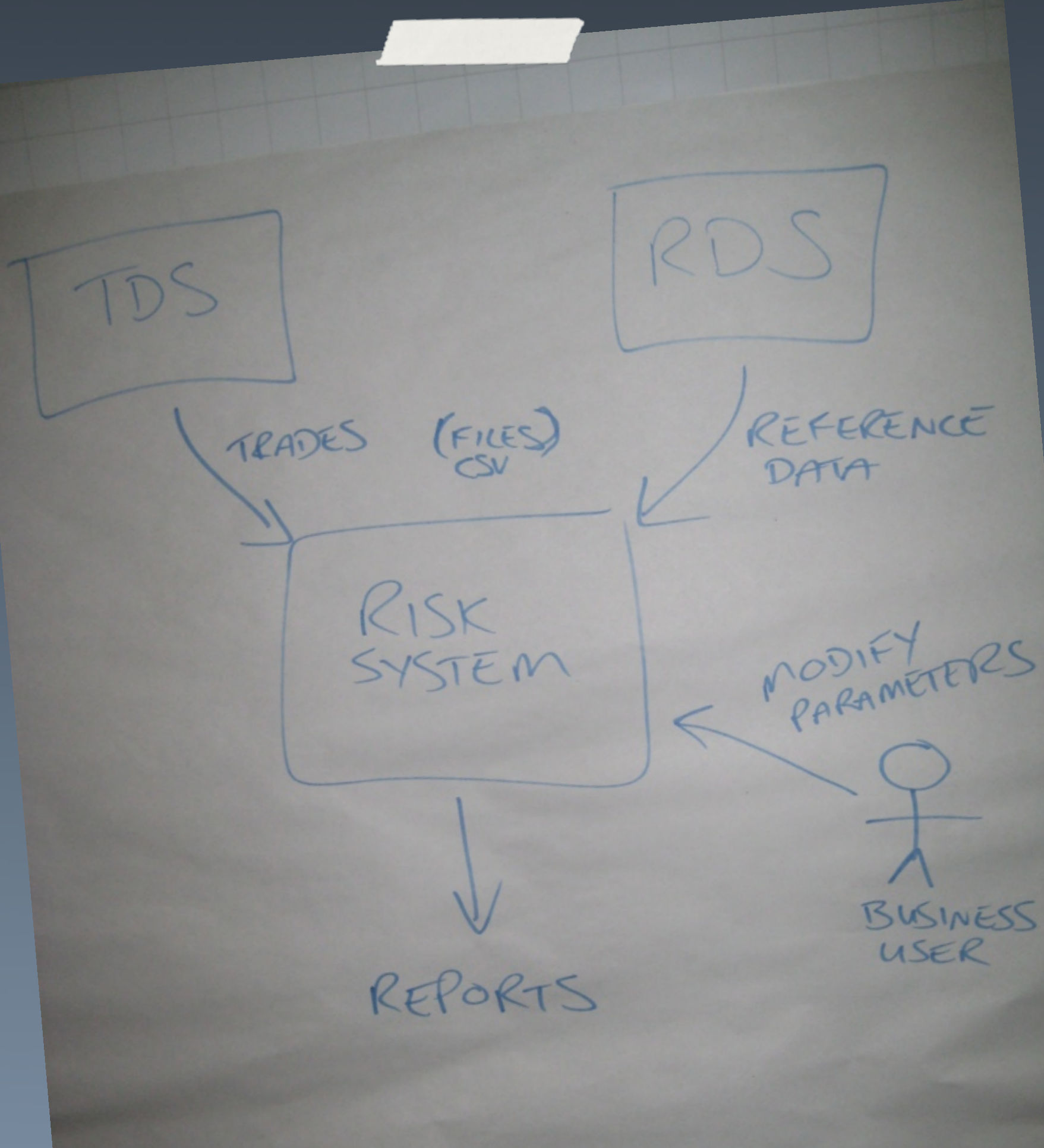
And we typically stick
to what we already

know

And therefore,

every problem can be solved with that technology

[Microsoft | Java | Open Source]


TDS
RDS
TRADES
(FILES)
CSV
REFERENCE
DATA
RISK
SYSTEM
MODIFY
PARAMETERS
BUSINESS
USER
REPORTS

Experience

is an essential
part of learning

Experience should guide,

not constrain

Successful software architecture is about

broadening the T

Use your experience to guide, but don’t let it constrain you

“Top 10 traits of a rockstar software engineer”

ReadWriteWeb

Top 10 Traits of a Rockstar Software Engineer

« Prior Post Next Post »

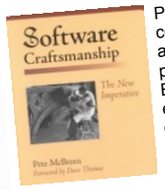
Written by Alex Iskold / April 8, 2008 12:50 AM / 40 Comments

Every company is a tech company these days. From software startups to hedge funds to pharmaceutical giants to big media, they're all increasingly in the business of software. Quality code has become not only a necessity, but a competitive differentiator. And as companies compete around software, the people who can make it happen - software engineers - are becoming increasingly important. But how do you spot the 'cream of the crop' programmers? In this post we outline the top ten traits of a rockstar developer.

We've written here before about the future of software development *(http://www.readwriteweb.com/archives/the_future_of_software_development.php)* , in which a few smart developers can leverage libraries and web services to build large-scale systems of unprecedented complexity. It only takes a couple of smart engineers to create quality software of immense value, and below is a list of the top ten qualities you should look for when hiring a developer:

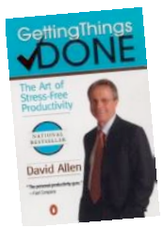
1. Loves To Code
2. Gets Things Done
3. Continuously Refactors Code
4. Uses Design Patterns
5. Writes Tests
6. Leverages Existing Code
7. Focuses on Usability
8. Writes Maintainable Code
9. Can Code in Any Language
10. Knows Basic Computer Science

1. Loves To Code



Programming is a labor of love. Like any occupation, truly great things are achieved only with passion. It is a common misconception that writing code is mechanical and purely scientific. In truth, the best software engineers are craftsman, bringing energy, ingenuity, and creativity to every line of code. Great engineers know when a small piece of code is shaping up perfectly and when the pieces of a large system start to fit together like a puzzle. Engineers who love to code derive pleasure from building software in much the same way a composer might feel ecstatic about finishing a symphony. It is that feeling of excitement and accomplishment that makes rockstar engineers love to code.

2. Gets Things Done



There are plenty of technical people out there who talk about software instead writing it. One of the most important traits of a great software engineer is that they actually code. They actually get things done. Smart people know that the best way to solve problems is go straight at them. Instead of spending weeks designing complex, unnecessary infrastructure and libraries, a good engineer should ask: What is the simplest path to solving the problem at hand? The recent methodologies for building software, called Agile practices *(http://en.wikipedia.org/wiki/Agile_software_development)* , focus on just that. The idea is to break complex projects into short iterations, each of which focuses on a small set of incremental features. Because each iteration takes just a few weeks to code, the features are manageable and simple. Teams that follow agile practices never create infrastructure for its own sake, instead they are focused on addressing a simple set of requirements. The secret is that when this approach is applied iteratively, a rich, complex piece of software arises naturally.

3. Continuously Refactors Code

Coding is very much like sculpting. Just like an artist is constantly perfecting his masterpiece, an engineer continuously reshapes his code to meet requirements in the best possible way. The discipline of reshaping code is known as refactoring *(http://www.refactoring.com)* and was formally described by Martin Fowler *(http://www.martinfowler.com)* in his seminal book *(http://books.google.com/books?id=1MsETFPD3I0C&dq=refactoring&ei=RrH3R8P5MpK2ygS-vum7Bw)* . The original idea behind refactoring was to improve code without changing what it does, moving pieces of the software around to ensure that the system is free of rot and also does what it is supposed to do based on current requirements. Continuous refactoring allows developers to solve another well-known problem - black box legacy code that no one wants to touch. For decades engineering culture dictated that you should not change the things that work. The issue, though, is that over time you become a slave to the old code, which grows unstable and incompatible. Refactoring changes that, because instead of the code owning you, you own the code. Refactoring establishes ongoing dialogue between the engineer and the code and leads to ownership, certainty, confidence, and stability in the system.

http://www.readwriteweb.com/archives/top_10_software_engineer_traits.php

1. Loves to Code (passion)

2. Gets Things Done
(actually writes useful code)

3. Continuously Refactors Code

(reorganises)

4. Uses Design Patterns

(spots opportunities to reuse experience)

5. Writes Tests
(confidence and proof)

6. Leverages Existing Code

(spots code reuse opportunities)

7. Focuses on Usability (working software)

8. Writes Maintainable Code

(clear and expressive)

9. Can Code in Any Language
(willing to learn)

10. Knows Basic Computer Science

(the basics)

Most importantly, rockstar engineers believe in ***simplicity*** *and common sense.*

11. Understands why (how and why does this work the way it does?)

These traits are the basis of a good **hands-on** software architect

The role of a hands-on software architect

coding {the} architecture

Why Software Projects Fail

www.codingthearchitecture.com

Simon Brown

Hands-on software architect

Detica

coding {the} architecture

It should be the architect

(somebody has to do it and that's why we get paid the big bucks)

Why software projects fail...

...architects are here to help, not to hinder

...-on software ...itect can be ...or preventing ...failure

Software projects fail for a number of reasons

Iterative and agile techniques solve *some* problems...

coding {the} architecture

The role of a software architect

Management of non-functional requirements

Architecture definition

Technology selection

Architecture evaluation

Architecture collaboration

Definition

Ownership of the bigger picture

Leadership

Coaching and mentoring

Quality assurance

Design, development and testing

Delivery

This is what we write about on our website...

www.codingthearchitecture.com

The software architect role is

more

than that of a lead developer

The software architect role is

different

to that of a lead developer

Software architecture is about

technical

and

soft skills

abstraction

Software architecture
is about a
different level of

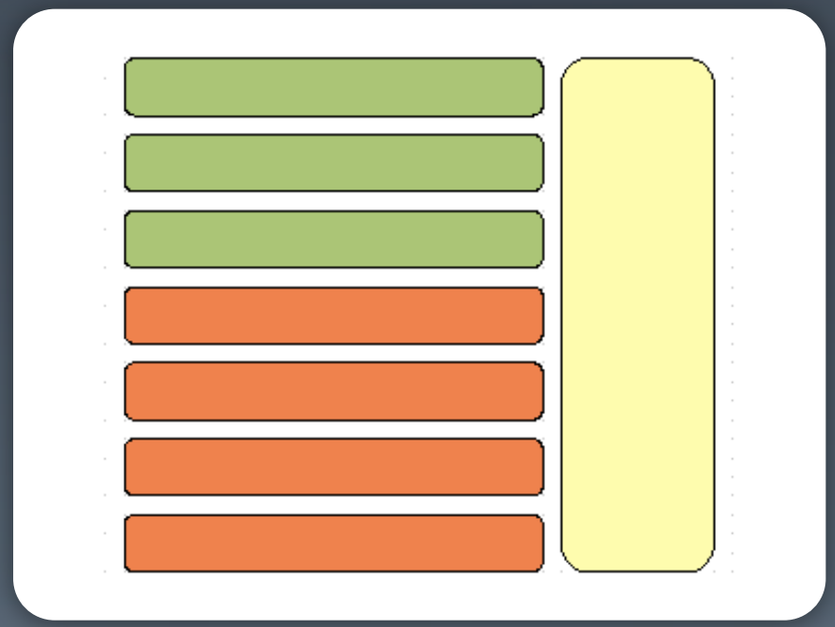
Abstract

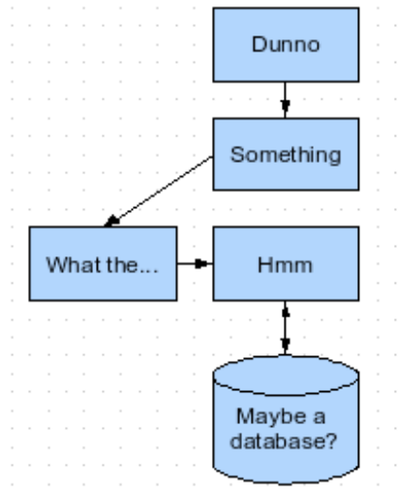

```
/// <summary>
/// Represents the behaviour behind the user registration page.
/// </summary>
public class RegistrationWizard : AbstractWizard
{
    private RegistrationCandidate _registrationCandidate;
    private RegistrationWizardPage _page;
    private RegistrationWizardController _controller;
```

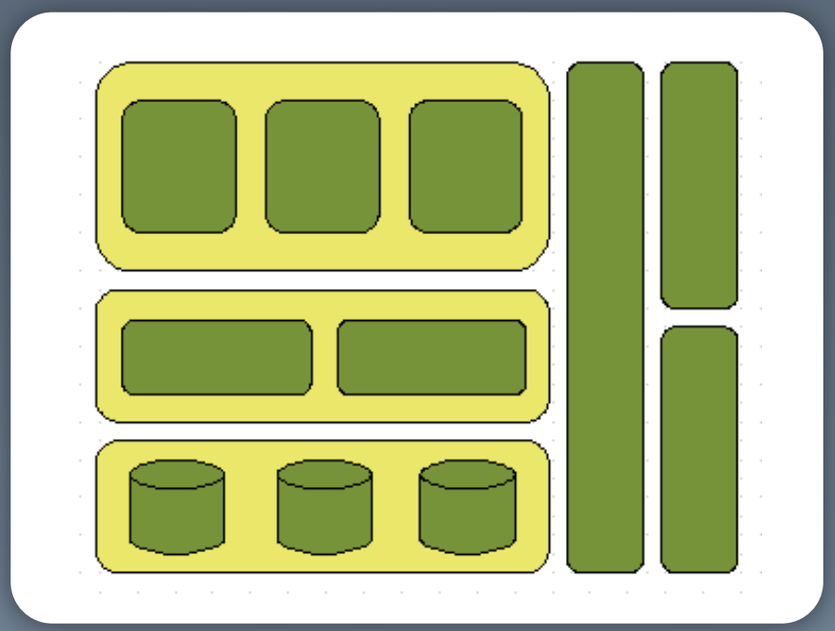
Specific

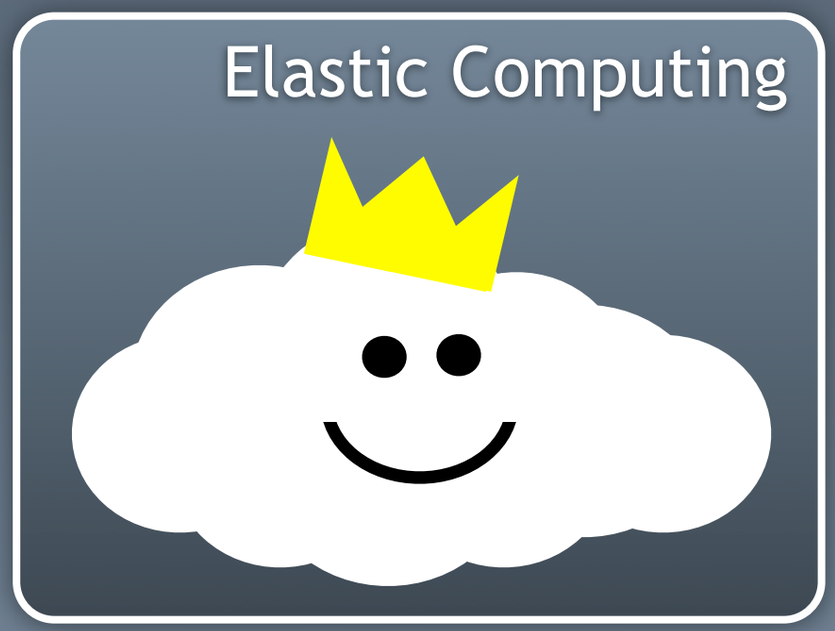
It’s about the

big picture




Dunno
Something
What the...
Hmm
Maybe a database?




Elastic Computing

It’s easy to be too

close

to the detail

Surely we mu
to simpli

is a messaging
ion?

syn

tter, my idea
tter.

S
A b

B
the-shelf

It's just nev
work that

t's make this the
master system...

STOP

They had lost sight of

the **big** picture

Learn about and understand

the **big** picture

Non-functional requirements

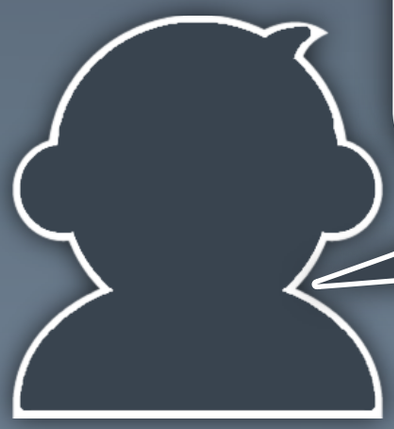
Non-functional requirements influence architecture

Good code != success

Typically teams struggle with

performance & scalability

“premature optimization is the root of all evil”



Donald Knuth

Most performance problems are

architectural

in nature

Too many

database

requests

Too many fine-grained network calls

(too chatty)

High memory consumption

(excessive allocation or garbage collection)

Use colocation to increase the overall performance

Colocation contradicts many scalability approaches

[loose coupling | partitioning | scaling out | ...]

Article

Scalability Principles

Posted by Simon Brown on May 21, 2008
Community Architecture Topics Performance & Scalability Tags Concurrency

At the simplest level, scalability is about doing more of something. This could be responding to more user requests, executing more work or handling more data. While designing software has its complexities, making that software capable of doing lots of work presents its own set of problems. This article presents some principles and guidelines for building scalable software systems.

1. Decrease processing time

One way to increase the amount of work that an application does is to decrease the time taken for individual work units to complete. For example, decreasing the amount of time required to process a user request means that you are able to handle more user requests in the same amount of time. Here are some examples of where this principle is appropriate and some possible realisation strategies.

- **Collocation** : reduce any overheads associated with fetching data required for a piece of work, by collocating the data and the code.
- **Caching** : if the data and the code can't be collocated, cache the data to reduce the overhead of fetching it over and over again.
- **Pooling** : reduce the overhead associated with using expensive resources by pooling them.
- **Parallelization** : decrease the time taken to complete a unit of work by decomposing the problem and parallelizing the individual steps.
- **Partitioning** : concentrate related processing as close together as possible, by partitioning the code and collocating related partitions.
- **Remoting** : reduce the amount of time spent accessing remote services by, for example, making the interfaces more coarse-grained. It's also worth remembering that remote vs local is an explicit design decision not a switch and to consider the first law of distributed computing – do not distribute your objects.

As software developers, we tend to introduce abstractions and layers where they are often not required. Yes, these concepts are great tools for decoupling software components, but they have a tendency to increase complexity and impact performance, particularly if you're converting between data representations at each layer. Therefore, the other way in which processing time can be minimised is to ensure that the abstractions aren't too abstract and that there's not too much layering. In addition, it's worth understanding the cost of runtime services that we take for granted because, unless they have a specific service level agreement, it's possible that these could end up being the bottlenecks in our applications.

2. Partition

Decreasing the processing time associated with a particular work unit will get you so far, but ultimately you'll need to scale out your system when you reach the limits of a single process deployment. In a typical web application, scaling out *could* be as easy as starting up additional web servers to handle the user requests and load balancing between them. What you might find, however, is that parts of your overall architecture will start to become points of contention because everything will get busy at the same time. A good example is a single database server sitting behind all those web servers. When that starts to become the bottleneck, you have to change your approach and one way to do this is to adopt a partitioning strategy. Put simply, this involves breaking up that single piece of the architecture into smaller more manageable chunks. Partitioning that single element into smaller chunks allows you to scale them out and this is exactly the technique that large sites such as eBay use to ensure that their architectures scale. Partitioning is a good solution, although you may find that you trade-off consistency.

As to how you partition your system, well that depends. Truly stateless components can simply be scaled out and the work load balanced between them, ideally with all instances of the component running in an active manner. If, on the other hand, there is state that needs to be maintained, you need to find a workload partitioning strategy that will allow you to have multiple instances of those stateful components, where each instance is responsible for a distinct subset of the work and/or data.

3. Scalability is about concurrency

Scalability is inherently about concurrency; after all, it's about doing more work at the same time. Technologies such as the early versions of Enterprise JavaBeans (EJB) attempted to provide a simplified programming model and encouraged us to write components that were single-threaded. Unfortunately, these components typically had dependencies on other components and this led to concurrency problems. If concurrency isn't thought about, you have systems where data can easily become corrupted. On the other hand, too many guards around concurrency lead to systems that are essentially serial in nature and limited in the degree to which they can scale. Concurrent programming isn't *that* hard to do, but there are some simple principles that can help when building scalable systems.

- If you do need to hold locks (e.g. local objects, database objects, etc), try to hold them for as little time as possible.
- Try to minimize contention of shared resources and try to take any contention off of the critical processing path (e.g. by scheduling work asynchronously).
- Any design for concurrency needs to be done up-front, so that it's well understood which resources can be shared safely and where potential scalability bottlenecks will be.

4. Requirements must be known

In order to build a successful software system, you need to know what your goals are and what you're aiming for. While the functional requirements are often well-known, it's the non-functional requirements (or system qualities) that are usually absent. If you do genuinely need to build a piece of software that is highly scalable, then you need to understand the following types of things up-front for the critical components/workflows.

- Target average and peak performance (i.e. response time, latency, etc).
- Target average and peak load (i.e. concurrent users, message volumes, etc).
- Acceptable limits for performance and scalability.

http://www.infoq.com/articles/scalability-principles

Understand

what non-functional requirements are all about

- [ ] Performance
- [x] Scalability
- [x] Availability
- [ ] Security
- [ ] Disaster Recovery
- [ ] Accessibility
- [ ] Monitoring
- [ ] Management
- [x] Auditability
- [ ] …

Runtime

- [ ] Flexibility
- [ ] Extensibility
- [ ] Maintainability
- [x] Interoperability
- [ ] Legal
- [x] Regulatory
- [ ] Compliance
- [ ] i18n
- [ ] L10n
- [ ] …

Non-runtime

Understand how to capture them

Understand how to refine them

Understand how to challenge them

Understand how they

influence

your architecture

Understand how to leverage technology for

Learn about and understand the non-functional requirements in order to build sufficient foundations

Patterns

Design patterns are

reusable

solutions to recurring problems

Patterns are a

communication

tool

[Singleton | Factory | Adapter | Proxy |
Builder | Facade | Filter | Flyweight | ...]

Patterns can also be found in the

big picture

n-tier

(physical and logical layering)

Service-oriented

Event-driven

Messaging

(point-to-point and publish-subscribe)

Pipes & filters

Tuple spaces

(distributed shared memory)

Grids

(compute and data nodes)

Cloud & elastic computing

Each pattern is suited to a certain

type

of problem

Synchronous vs asynchronous

Performance

vs scalability

Open vs closed

Security

Availability & failover

Each pattern has its own set of trade-offs

Understand each from

the **big**

picture perspective…

...while understanding

how they

work

at the code level

Adapt

the patterns to meet your needs

Look at past projects, identify the patterns in use and understand why they were chosen

Testing

Testing provides confidence

Most people are comfortable with functional testing techniques

What about testing the

non-functional

requirements?

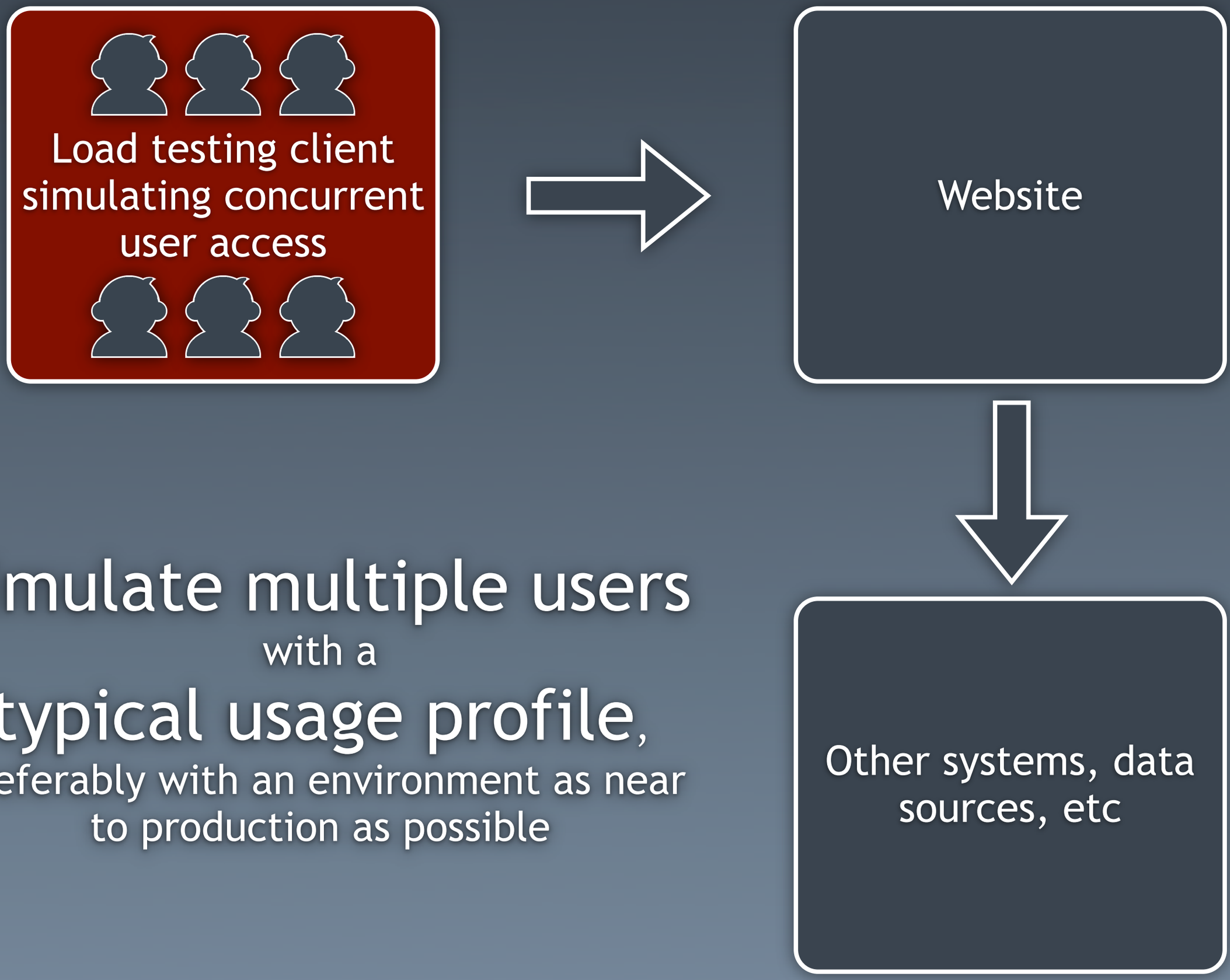
Load testing is one way to evaluate your architecture

(if performance and/or scalability is important to you)

Load testing: asserting how the architecture performs under load with a view to monitoring the response times for key transactions.

Soak testing: asserting that the performance of the architecture remains stable over longer periods of time.

Stress testing: asserting what the upper bounds are for the scalability of the architecture, understanding how it reacts when stressed.



Simulate multiple users

with a

typical usage profile,

preferably with an environment as near to production as possible

1. Understand the non-functional requirements

Performance and scalability characteristics, typical usage profiles, etc.

2. Define the test script

Determine the actions to simulate from the test script and implement with a load testing tool.

3. Schedule and configure environment

Book testing slots to minimise disruption and configure data for load testing.

4. Determine metrics to record and monitor

Ensure that the test script captures the appropriate statistics and determine what system characteristics to monitor (e.g. CPU, RAM, IO, etc).

5. Execute pre-tests

Test the test scripts and monitor, refining if necessary.

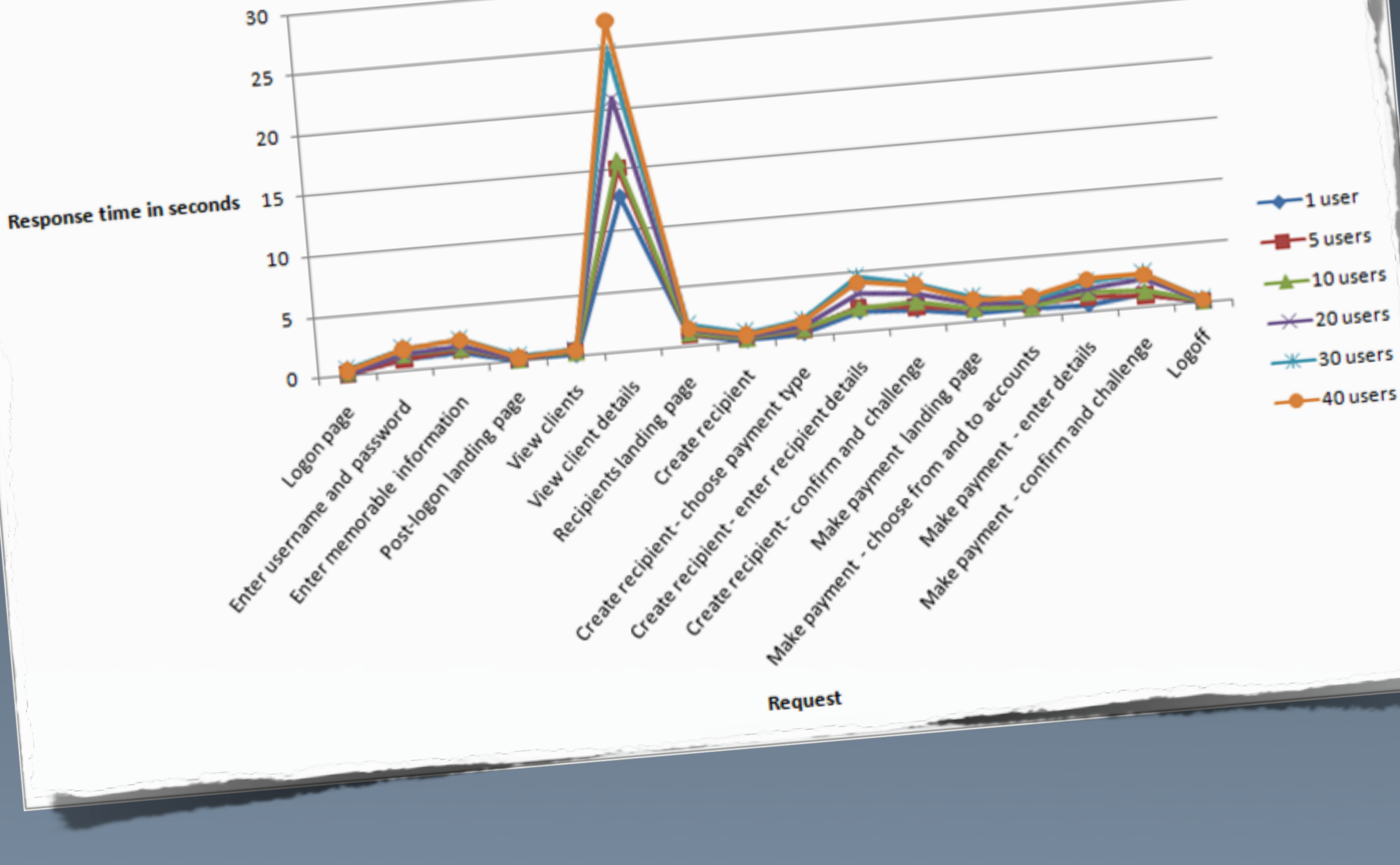
6. Execute tests

Clear down the environment, warm it up, execute tests at varying levels of concurrent usage.

7. Analyse results

Calculate useful statistics (average, maximum and 95th percentile response times), draw graphs and make conclusions.

Average response times (raw)



What happens if you find

performance problems?

Look at your monitoring data

and log files,

run profilers,

etc…

Maybe the environment needs tuning

[Memory | Connection pools | Worker thread pools | Database indexes & hints | etc]

The **big** picture and

the **detail**

are equally as important

Understand how to test, monitor, diagnose problems and tune your systems

Summary

is for technology

Deep hands-on
technology skills

Experience should guide,

not constrain

The software architect role is

different

to that of a lead developer

It’s about the

big picture

But the

detail

is equally as important

The top 10 traits of a rockstar software engineer are applicable to architecture

Deep hands-on technology skills

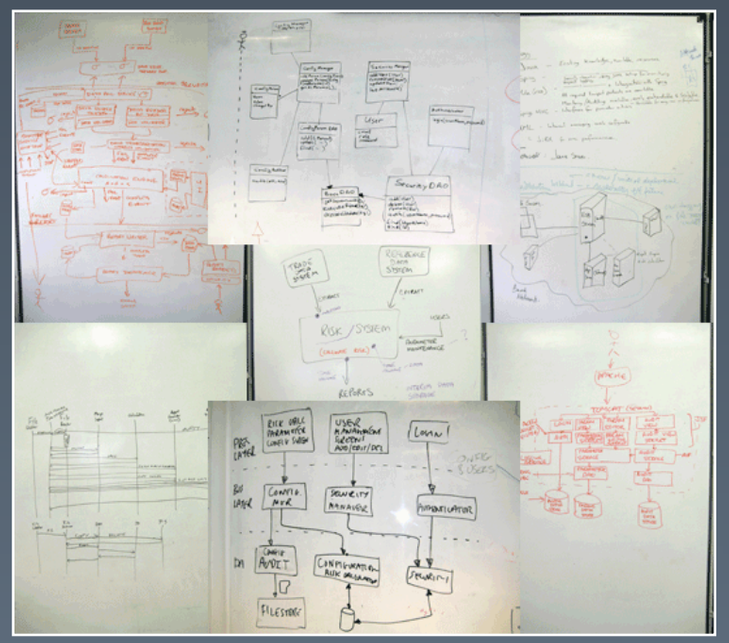
Broad knowledge
of patterns, designs,
technologies ...
options & trade-offs

Deep hands-on
technology skills

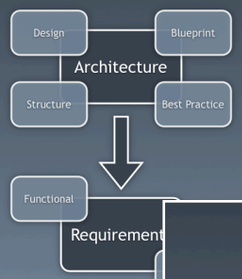
From Developer to Architect

A 2-day interactive introduction to software architecture

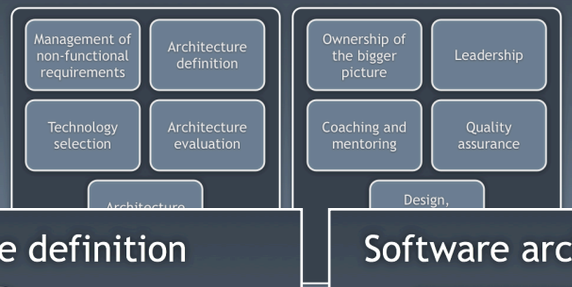
Our “From Developer to Architect” training course is about broadening your software development skills and has been designed to take full advantage of the technical knowledge that you already have; whether that’s .NET, Java or something else. This course will make you more “architecturally aware”, helping you build better software. It’s about pragmatic and real-world software architecture rather than academic “ivory tower” software architecture.



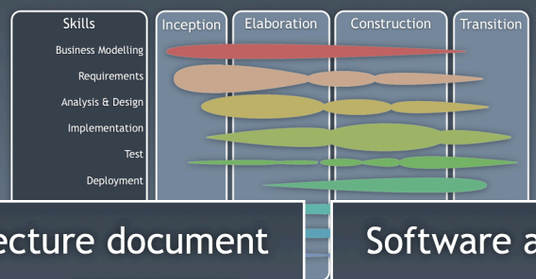
What is architecture?



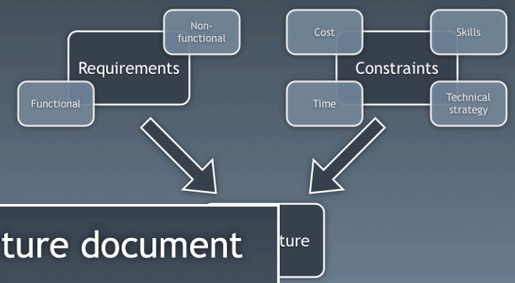
The role of a software architect



Architecture & RUP

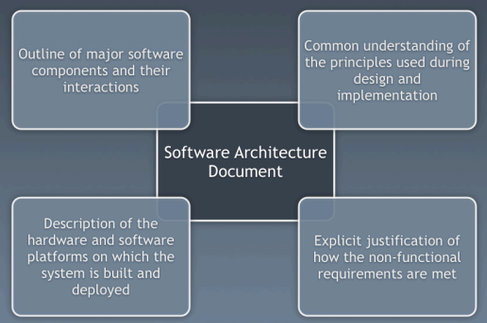


Architectural drivers

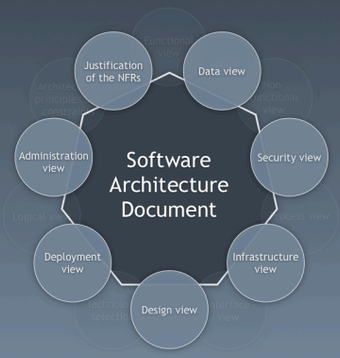


Architecture definition

Software architecture document



Software architecture document



coding {the} architecture

"Coding the Architecture" is a website and community for hands-on, pragmatic software architects. You'll find content and discussion about architecture and the role of an architect, along with our experiences of undertaking that role.

E-mail: simon.brown@codingthearchitecture.com
Website: http://www.codingthearchitecture.com
Google Group: http://groups.google.com/codingthearchitecture