Top Ten Software Architecture Mistakes

Eoin Woods
Barclays Global Investors

www.eoinwoods.info
www.barclaysglobal.com/careers

Content

- Introduction
- Ten Mistakes and Some Solutions
- Recap & Conclusions

Introduction for Me

- Software architect at Barclays Global Investors
  - responsible for Apex, a new portfolio management system for Active Equity
  - also involved in regional technology architecture, trading systems and global architecture council
- Software architect for ~10 years
  - with some enterprise architecture for about 2 years
- Author of "*Software Systems Architecture*" book with Nick Rozanski
- IASA and BCS Fellow, IET member, CEng

Introduction for the Talk

- Based on an article written for IT Architect
  - itarchitect.co.uk
  - commissioned through IASA (www.iasahome.org)
- Ten mistakes that I've made and seen made
  - ten is an arbitrary number, but a good size to start with
  - of course there are others, your top 10 may be different
- Most are simple but they happen again and again
  - solutions are also quite simple but have been effective
  - I've found that simplicity usually means effectiveness

Content

- Introduction
- Ten Mistakes and Some Solutions
- Recap & Conclusions

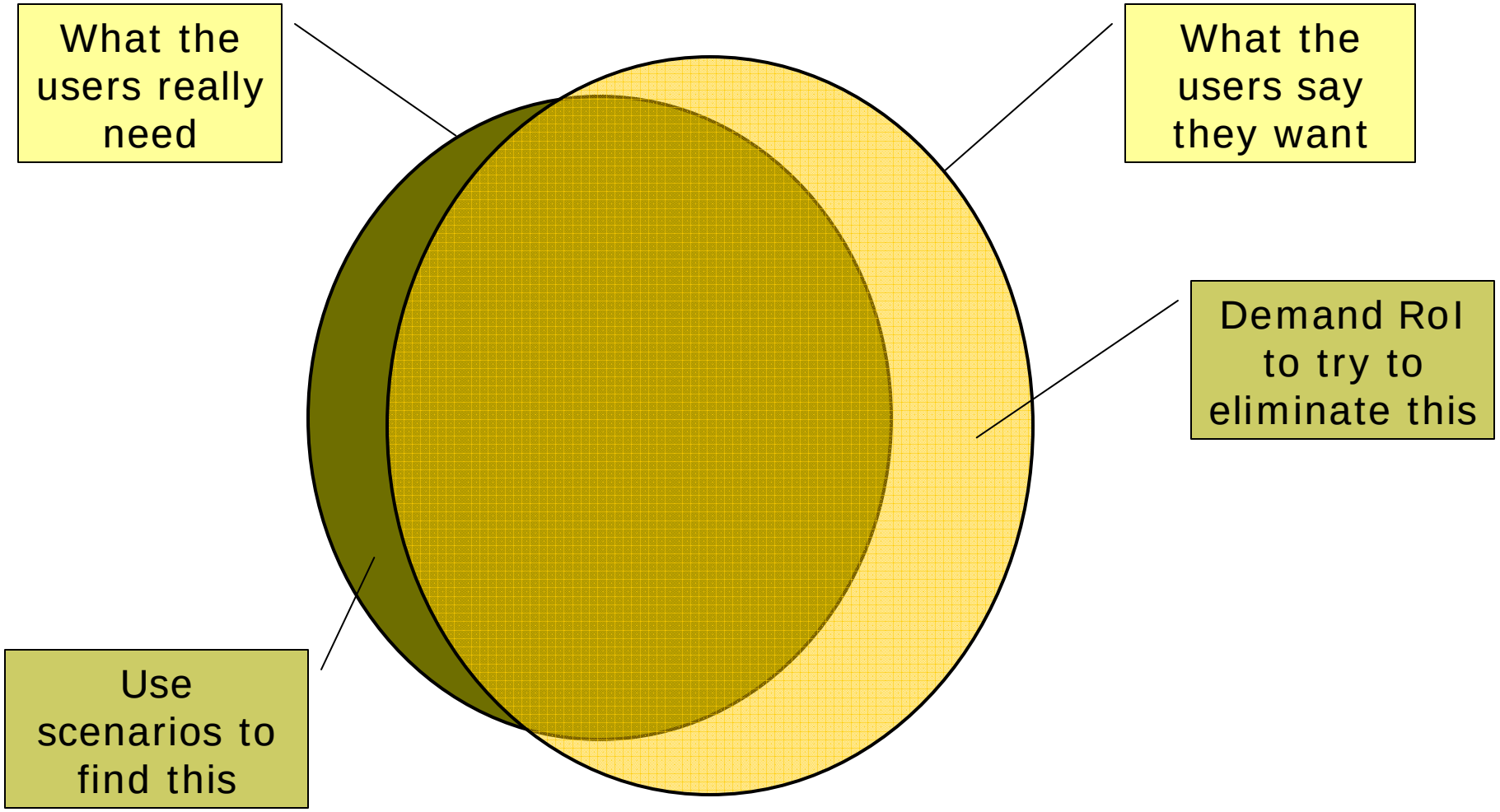
Mistake 1: Scoping Woes

- Scope creep is the enemy the delivery deadline
  - expense systems that end up processing invoices!
  - but too narrow a scope can be as bad as too broad
- Functional scope problems are well understood
  - “show me the money” – why is this important to you?
  - “missing bricks” – the boring pieces the rest need
- Non functional scope mistakes are more difficult
  - “available 24 x 7 x 365” … or Mon-Sat 0800-2000 ?
  - “Usability of the Mac” … or perhaps better than Access ?
  - “No authorisation needed” …. do *anything* after login ?

Solution 1: Controlling Scope

- Focus ruthlessly on the problem being solved
  - usually needs deep domain knowledge
  - get help, ask for a range of independent opinions
- Always consider your system in the larger context
  - does it solve a complete problem in *your* environment?
- If asked to gold plate challenge the RoI
  - "how will this feature increase your effectiveness?"
- If features are missing illustrate with scenarios
  - show why it won't work without the missing pieces
  - may need to trade off manual business process steps

When Scopes Collide



Mistake 2: Not Casting Your Net Widely

- Systems are built to meet stakeholder needs
  - everything ultimately tied to a stakeholder who cares
- The difficulty is working out who is important
  - Users ✓
  - Acquirers (budget holders) ✓
  - Support staff and systems administrators *?*
  - Vendors *?*
- Include everyone whose cooperation is needed
  - IT Security, IT Risk, Compliance?
  - IT Operations?
- Consider positive *and* negative stakeholders

Solution 2: Building a Stakeholder Group

- Need a stakeholder list as early as possible
  - even if not formal, *you* need to understand it early
- Consider who is affected by the system
  - individuals and groups
  - positive and negative
- Rank by influence and likelihood of disagreement
  - who *really* cares one way or another
  - how likely are they to create problems!
- Get to work on the (H:H) people immediately!
  - use the ranking to prioritise communication
  - earlier communication generally reduces problems

Solution 2: Example Rankings

- Acquirer – sponsors the project
  - probably medium or high interest but low risk (M:L)
- End Users – use whatever you build
  - probably high interest, low risk if involved (H:L)
- Compliance – concerned about legal regulation
  - probably medium interest, medium risk (M:M)
- IT Security – concerned about standards & risk
  - probably medium interest, high risk (M:H)
- IT Infrastructure – concerned about running it
  - probably high interest, medium risk (H:M)

Stakeholder Groups

- *Acquirers* pay for the system
- *Assessors* check it for compliance
- *Communicators* create documents and training
- *Developers* create it
- *Maintainers* evolve and fix the system
- *Suppliers* provide system components
- *Support Staff* help people to use the system
- *System Administrators* keep it running
- *Testers* verify that it works
- *Users* have to use the system directly

Mistake 3: Focusing on Function

- End user cares what the system does
  - but actually cares how it does it too
  - slow, difficult to use, unavailable … won't get used
  - other groups even more so (e.g. supportability)
- Easy to forget everything apart from functionality
  - "when will I be able to do X?"
  - rarely ask "… and how fast will that be?"
- Good design makes functions "easy" to add
  - although data is often another question … ☺
- Qualities are often difficult to change later
  - expensive to add security, availability, performance, …

Solution 3: Consider Your Qualities

- Work through the standard list
  - availability, compliance, evolvability, maintainability, performance, reliability, scalability, security, supportability
- Pick your top 3 or 4
  - you can't deal with them all at once
  - select by product of importance and difficulty
- Identify requirements & technical solutions
  - perspectives can help in new areas
- Identify the conflicts between them and trade-off
  - this is the hard (but interesting) part

Solution 3: Example Trade-off

- Your system requirements state that it has to be “secure”
  - sensitive operations and/or data
- You consider threats and risks and decide on
  - client PKI certificates and two factor authentication
  - role-based access control w/ information partitioning
  - prevent leakage (no USB keys, no copy & paste, …)
- Secure system that meets your written requirements
  - but will anyone be prepared to use this system?
  - how expensive will build, operation and support be?
- So what do you decide to trade-off?
  - there’s no “right” answer to this
  - example may be fine for a military system, not for call centres

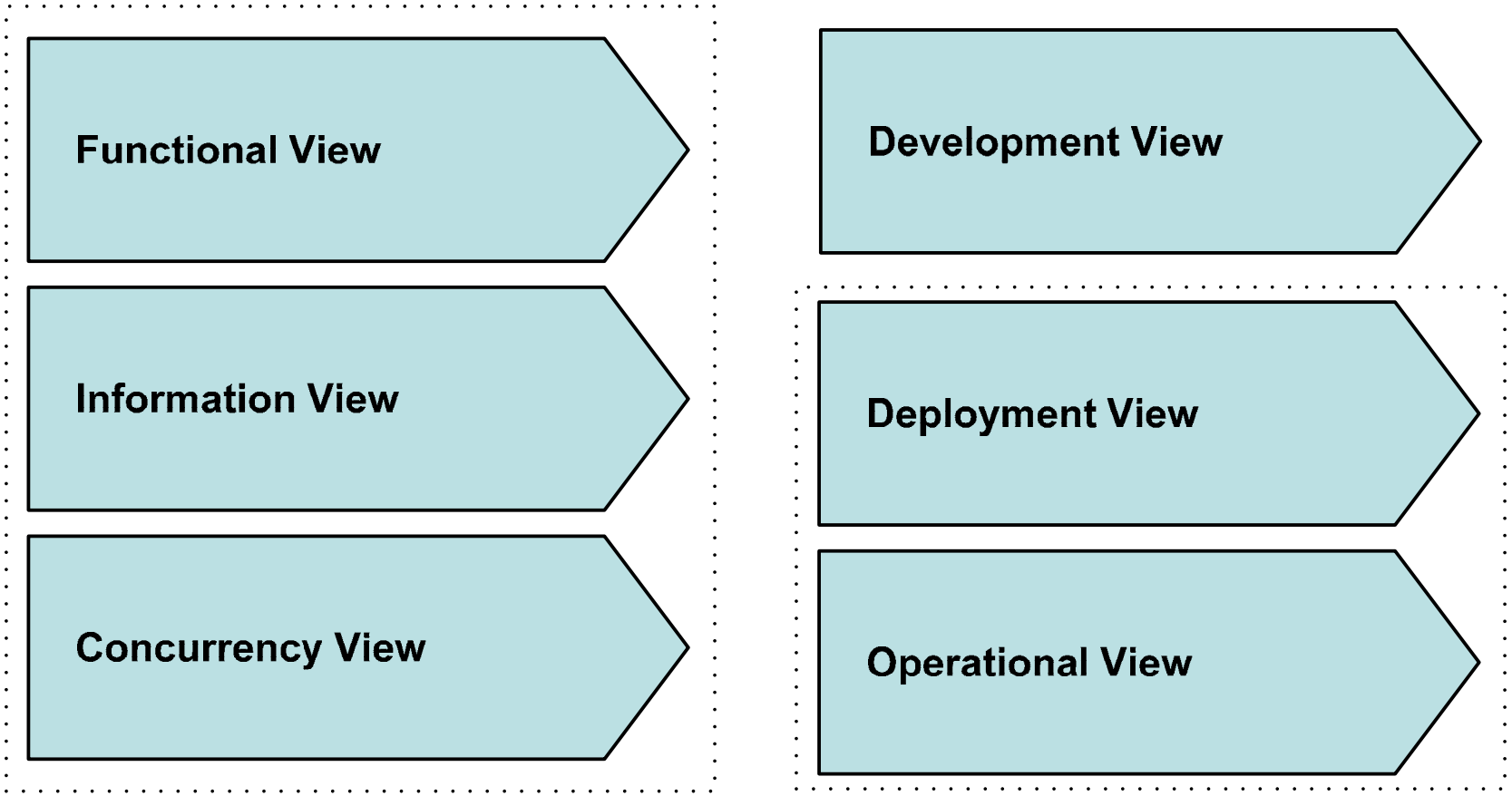
Mistake 4: Boxes and Line Descriptions

- Communicating your architecture is crucial
  - if no one understands then the architecture won't exist
- Difficult to represent architecture on paper
  - complicated, large, multi-faceted, subtle, ...
- Different people care about different things
  - DBAs – database, data location, data usage, ...
  - IT Infra – machines, connections, middleware, ...
- One large description rarely works well
  - so consider views of your architecture
- A badly defined description never works well
  - so don't use PowerPoint/Visio boxes and lines!

Solution 4: Adding Precision to Description

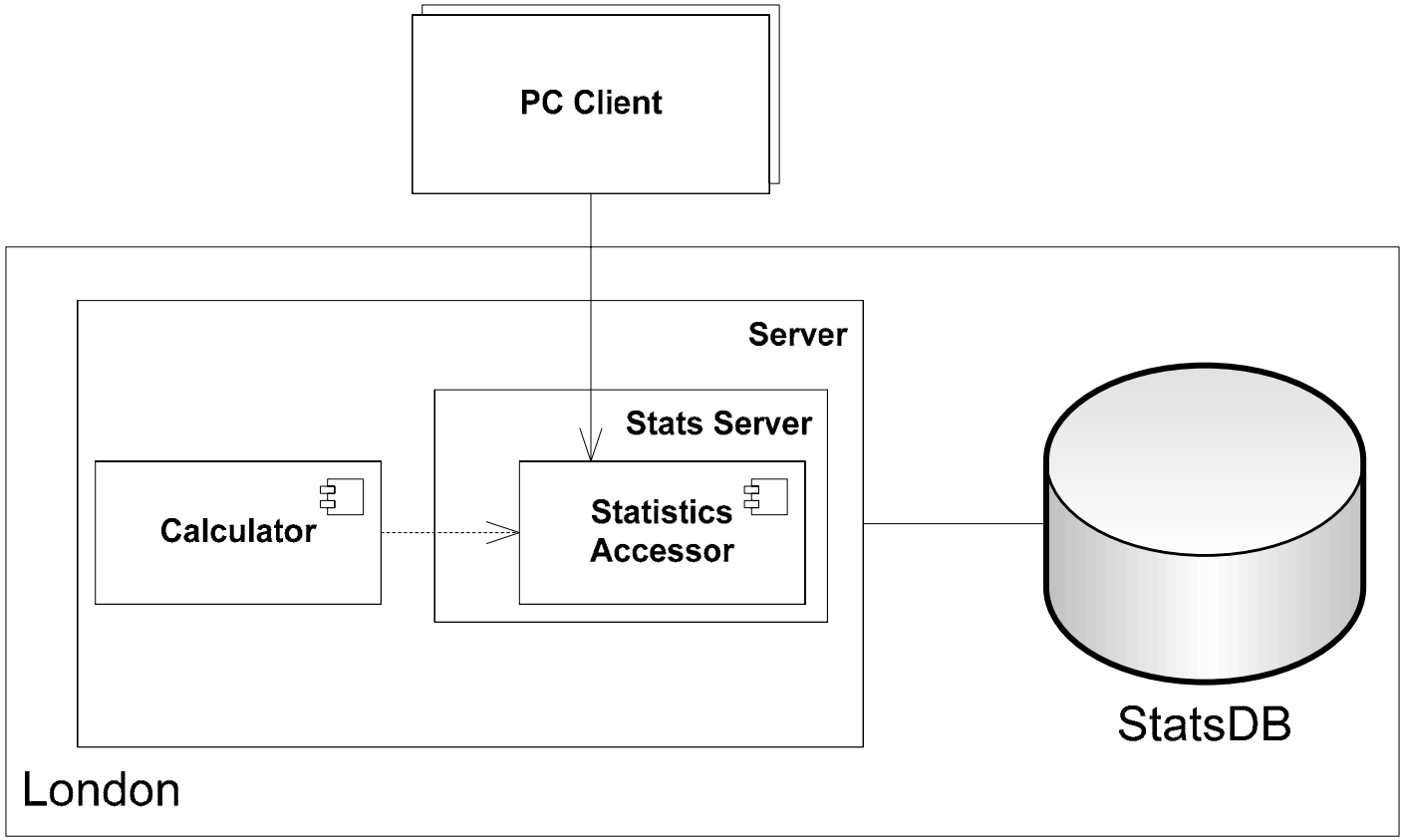
- USE A WELL DEFINED NOTATION
  - if necessary use a very simple one
  - otherwise everyone will read it their own way
- Using UML? Define conventions and stereotypes
  - so what exactly is a “component” then?
- Break your description down into views
  - one view per type of structure
  - functional, concurrency, deployment, information, …
- Be accurate, even when abstract
  - suppressing detail shouldn’t mean introducing errors!
  - imprecise descriptions are confusing (“but I thought …”)

Solution 4: Example View Set



From “Software System Architecture”, Rozanski and Woods, 2005

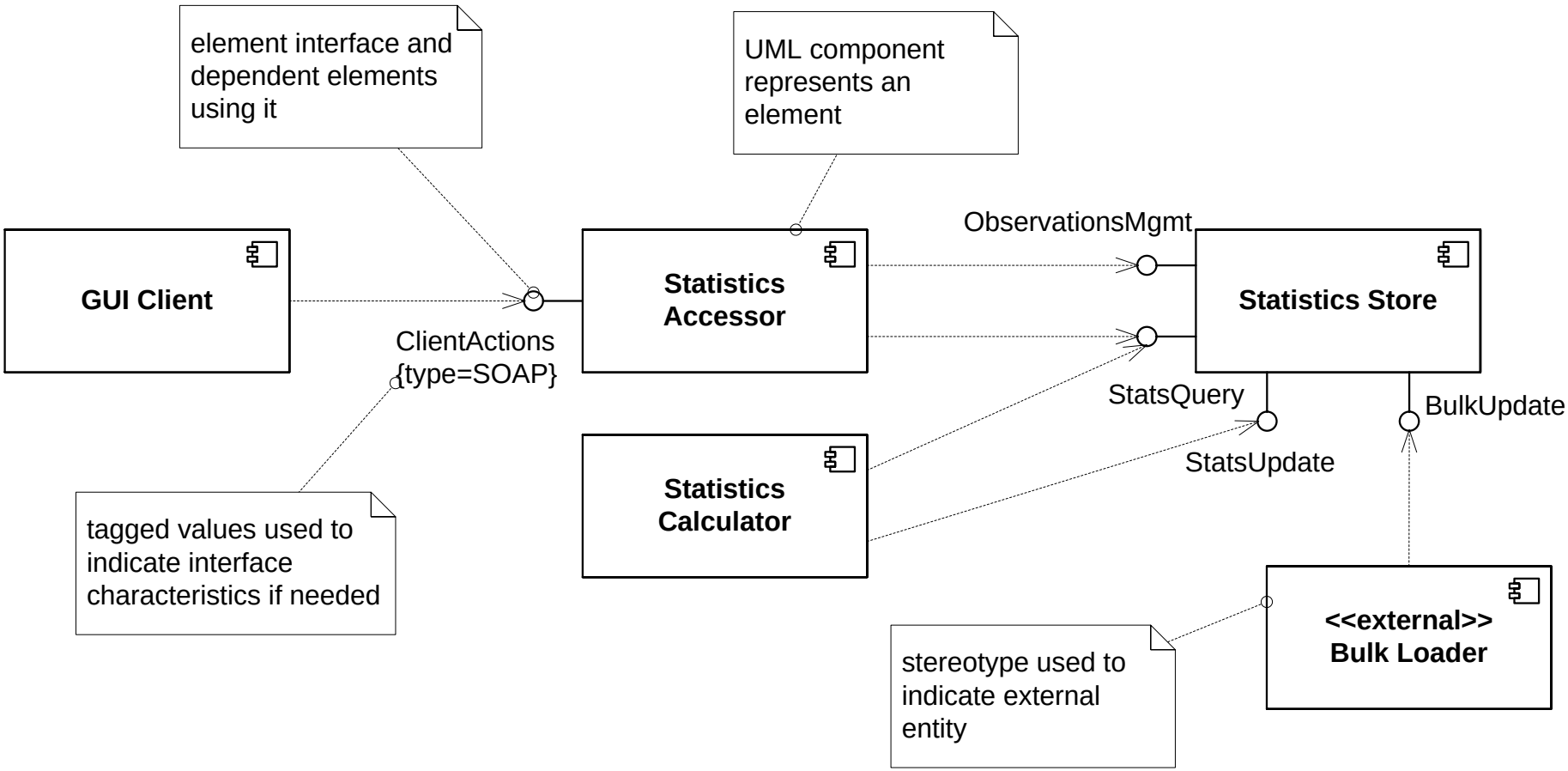
Solution 4: Example of Confusion



We *probably* understand this but it's hard to be sure. What does all of the notation mean? Are all the relationships there?

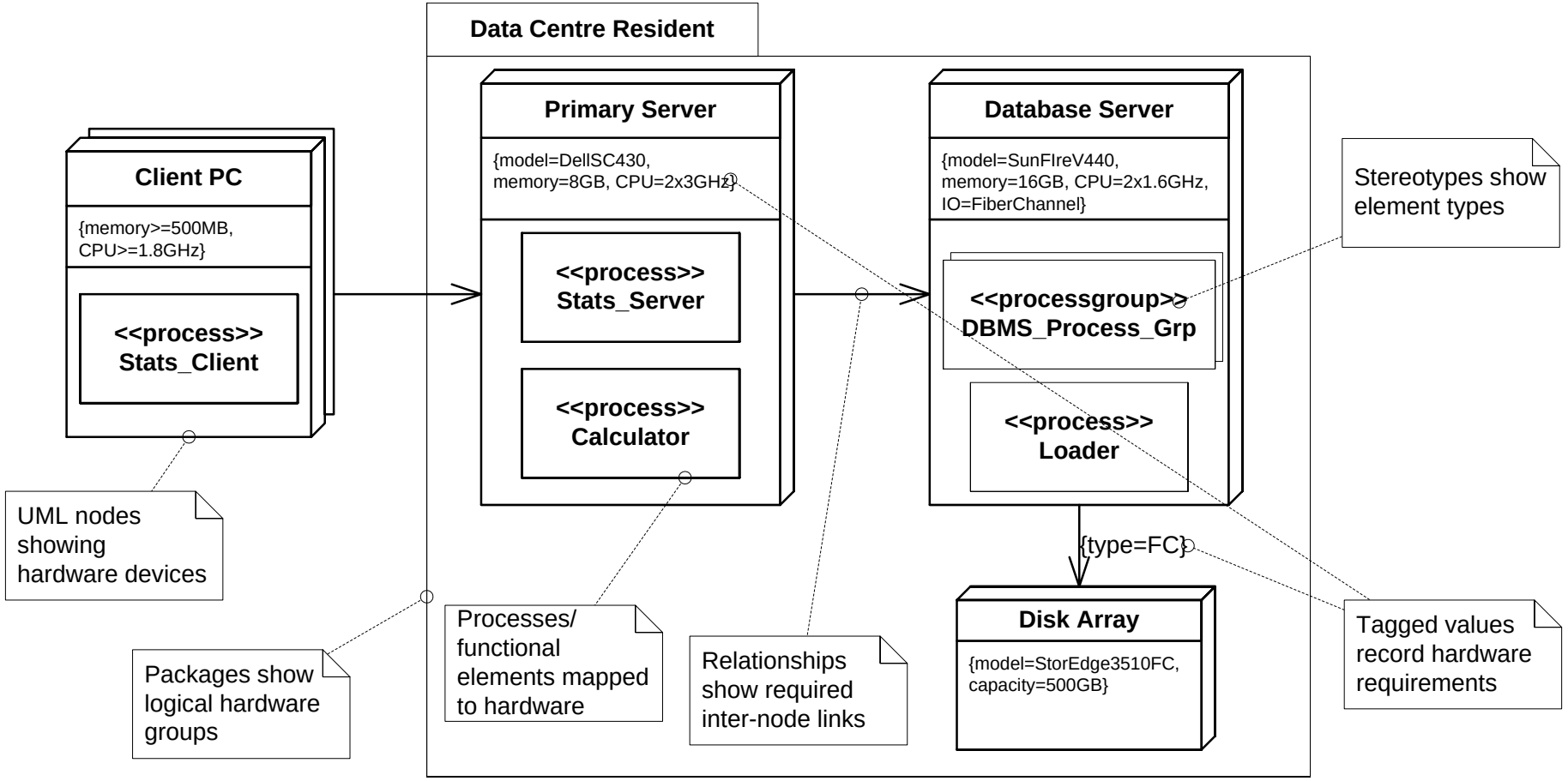
Solution 4: Better Example

A Functional View Fragment



Solution 4: Better Example

A Deployment View Fragment



Mistake 5: Forgetting It Needs to be Built

- Everyone wants to do new and cool stuff
  - COBOL/VSAM vs. Ruby on Rails? Any takers for COBOL?
- Most systems could be built in a variety of styles
  - file transfer or message based events or SOA?
- Using a range of technical options
  - single database vs. distributed database?
  - full app server vs. simple framework use?
- Every option has its own dependencies & risks
  - Can your design be realised at acceptable risk and cost in *your* environment?
  - people, time, budget, experience, ... all have an impact

Solution 5: Grounding Your Architecture

- Consider what it takes to build your architecture
  - do you have the people who can do it?
  - do you have the budget and time?
  - do your acquirers have the risk tolerance?
  - can the organisation deal with it? (e.g. Infrastructure)
- Are the innovations beneficial enough to justify?
  - is the added performance really needed?
  - or the flexibility likely to be used?
  - can the reduced time to market be capitalised upon?
- Could you achieve the same thing in other ways?
  - using existing approaches in new/better/different ways?

Solution 5: Grounding SOA

SOA Benefit	Alternative Approach
Loosely Coupled	Messaging, ETL, common file formats, REST, …
Cross Platform	VM-based languages, XML, HTTP, even flat files !
Reusable Components	Interfaces, component models, architectural style, …
Process Flexibility	Traditional workflow, configurable systems (e.g. via Spring)
Buzzword Compliant	REST or Functional languages ☺

This is not to say that SOA doesn't have value, but many of its asserted benefits can be provided in other (maybe simpler) ways

Mistake 6: Lack of Platform Precision

- Modern systems rely on a stack of dependencies
  - virtual machines, operating systems, middleware, containers, XML libraries, database libraries, database systems, GUI toolkits, ...
  - A potential versioning nightmare
- Packaging dependencies with the application minimises version problems
  - but can't package the OS, database, ...
- Easy to be quite imprecise about dependencies
  - "Runs on Linux, WebLogic 10 and Oracle"
  - chances of this ending well are slight
- Real risks if dependencies aren't understood

Solution 6: Specifying Your Platform

- The easy solution: inventory your platform
  - standardise your environments
  - write down *exactly* what you use
  - impose it on your support group (or self support)
- The better solution: use standard builds
  - use the “approved” software “stack” in your organisation
  - stay current during development and test
  - less flexible & only works if there are pre-built stacks!
- The best solution: interoperability testing
  - also the most expensive solution
  - allows a range of platform options to be used
  - probably only appropriate for products

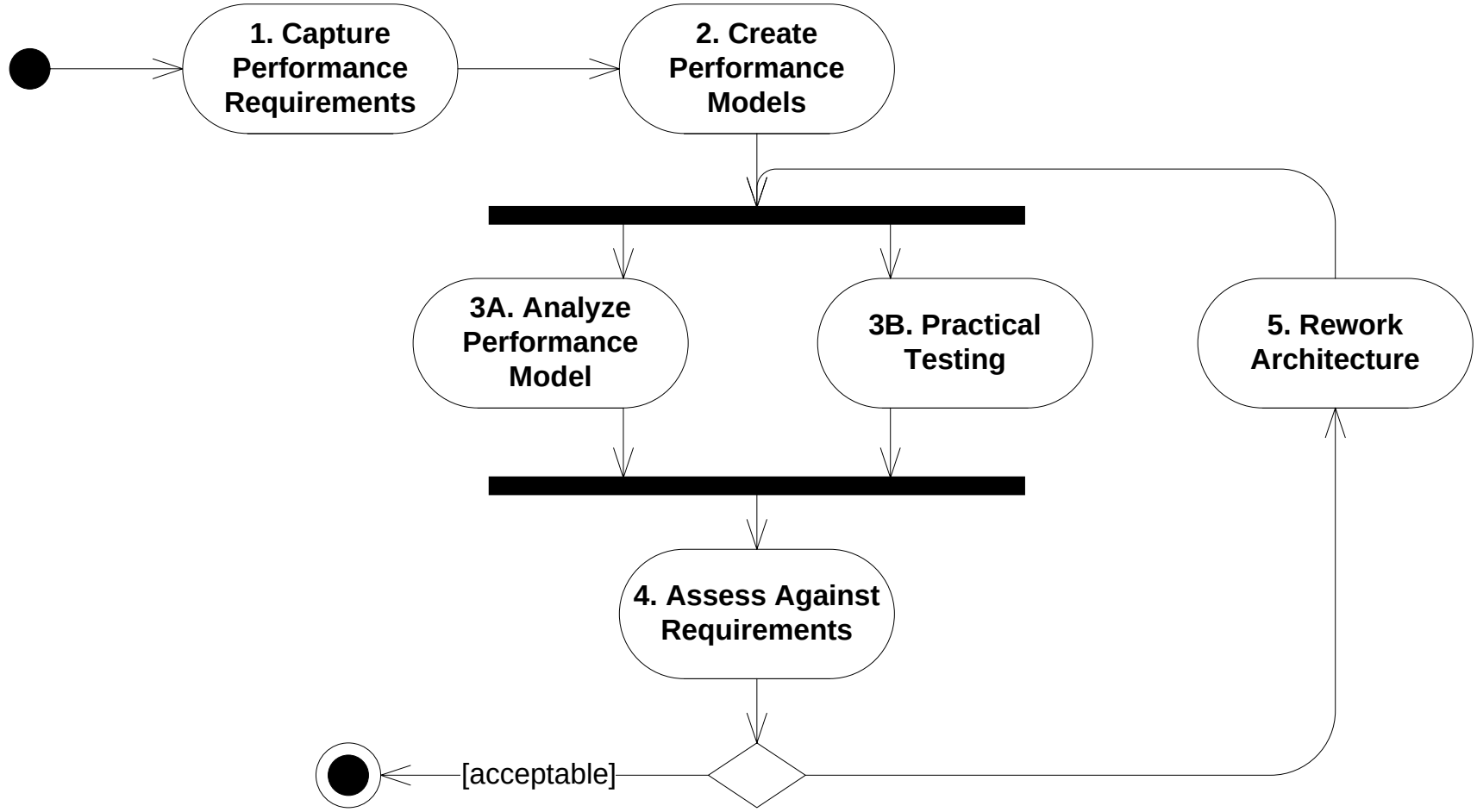
Mistake 7: Performance Assumptions

- Performance is a hard quality to guarantee
  - and unfortunately nearly everyone is interested in it!
  - often dependent on environmental factors and obscure technical details (as well as your design of course)
  - difficult and expensive to test thoroughly and reliably
- Easy and tempting to assume all will be well
  - test something small, multiply the answer!
  - this hardly ever works
- Performance needs to be tackled early
  - estimation and modelling
  - testing in the small
  - testing in the large

Solution 7: Assume Nothing!

- Performance is one example of a difficult quality
  - plenty more: security, scalability, evolution, ….
- Assuming anything about qualities is a mistake
  - all of them depend on a range of subtle factors
- The strategies to mitigate the risk:
  - use the experience of experts
  - review assumptions and designs widely
  - test with prototypes and test beds
  - model qualities to allow “pen and paper” analysis
- Practical testing is usually the most effective approach
  - but expensive in $$ and time
  - you still need to consider interaction of factors

Solution 7: Dealing with Performance



From the R&W "*performance perspective*" - aims to provide a guide to avoiding performance surprises late in the day

Mistake 8: DIY Security

- Conceptually security isn't that hard
  - do I know you? can you do or see this?
- In reality security is very easy to get wrong
  - vulnerability to analysis, replay or just guessing
  - gaps in protection, covert channels for information
  - subtle inconsistencies leading to vulnerabilities
- Security technology is often complicated
  - "easier if I just build something – less risk"
  - difficulties emerge in operation, breach or assessment
- Try to use standard solutions in standard ways
  - otherwise get expert, experienced help

Solution 8: Reuse Infrastructure

- Security is just one example of this concern
  - high performance, scalable servers aren't easy either!
- In general reusing infrastructure is safer
  - it's been written already so you know what you get
  - it comes at a cost you can estimate
  - you can test it to see if it works
  - it *probably* has a lot of the problems ironed out
- But as ever it's a trade off to make
  - generic products don't solve *your* problem specifically
  - they can introduce a lot of complexity and unknowns
  - can introduce a lot of initial adoption cost too

Solution 8: Reusing Infrastructure

- Using an authorisation package could be complex
  - cost, deployment complexity, integration complexity, runtime dependencies, availability risks
- Authorisation just needs a (role, action, resource) table and code to check it doesn't it?
  - but it also needs administration interfaces ...
  - and integration into your security processes ...
  - and auditing of all changes ...
  - and must be secure and tamper proof ...
  - and ...
- Actually, maybe a package isn't such a bad idea!
  - if it provides everything you need reliably of course

Mistake 9: Lack of Disaster Recovery

- Early work on disaster recovery seems unnecessary
  - "No time to worry about things that may never happen"
- Unfortunately that's rarely the case
  - internal requirements (risk to the business)
  - external compliance (SOX, OCC, FSA, SEC, ...)
- DR is expensive and complicated
  - cost of the DR environment and process
  - recovery to a DR environment never goes well initially
  - running regular realistic tests is the only way to check it
- Early work on DR design is the only way to get there
  - making sure the system *could* be recovered
  - getting the dependencies set up

Solution 9: Practice, Practice, Practice

- Plan, Design, Build, *Practice*
- Starting planning for availability early
  - can HA mitigate some of the DR situations?
  - budget for the time and money needed (no surprises)
- Put DR in the design and build work
  - review your designs for disaster recovery difficulties
  - allow for the geographical distribution needed
  - consider how you'll deal with data loss and latency
- Practice, practice, practice
  - run reasonably representative recovery exercises
  - start as soon as there is a system running

Mistake 10: No Backout Plan

- We always hope that deployment will go well
  - but we've all experienced situations when they don't
  - many factors causing failure are outside your control
- A backout plan deals with failed deployment
  - detailed concrete steps for restoring the status quo
- Easy to ignore or skimp on a backout plan
  - again, an unnecessary luxury you can ill afford time for
- Without one you risk total unavailability
  - "if you think education is expensive try ignorance"
  - rare to find an application where this is a good trade

Solution 10: Know Where You Came From

- Reality means that upgrades go wrong
  - unexpected environment, infra faults, system faults, …
- Failed upgrades mustn't affect availability
  - but often have to use the same hardware and databases
- Upgrades need reverse gear as well as forward
  - At any point you need to be able to back out (or be clear you can't and have a contingency)
- For large systems this is often difficult
  - parallel hardware or databases? and networking, … ?
  - handling workload during long upgrade windows?
  - dealing with multiple component versions concurrently
  - no magic formula – it needs thought, ingenuity & diligence

Content

- Introduction
- Ten Mistakes and Some Solutions
- Recap & Conclusions

Recap

- Scoping Woes
- Not Casting Your (Stakeholder) Net Widely
- Focusing on Functions (Forgetting Qualities)
- Using Box and Line Descriptions
- Forgetting that it Needs to be Built
- Lack of Platform Precision
- Performance Assumptions
- DIY Security
- Lack of Disaster Recovery
- No Backout Plan

Architecture Effectiveness Index

- Level 0 – no architectural impact
  - Architecture largely ignored, seen as irrelevant
- Level 1 – Stopping things getting worse
  - Essential decisions coordinated
- Level 2 – Stable and organised
  - Architecture understood, shared, aids change
- Level 3 – Architecture centric
  - Architecture is the point of reference for change

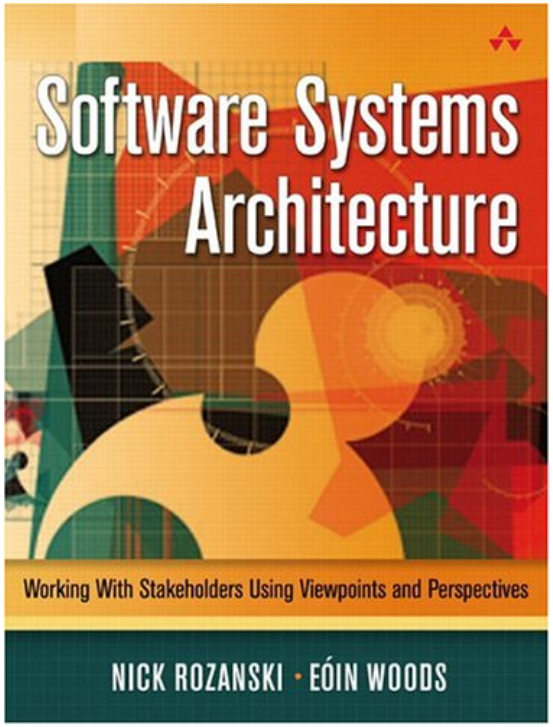
Summary

- No one ever said software architecture was easy
  - but there are mistakes that get made again and again
  - some have well known solutions, some don't
- Just an awareness often helps
  - many are related to maintaining a broad view
  - software architecture is more than module design
- Broad technical knowledge is valuable
  - ability to deal across specialisations is key to the role
- Risk and return approach is key
  - what makes the system more valuable?
  - what is likely to cause it to fail?

A Few More Solutions

Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives

Nick Rozanski & Eoin Woods
Addison Wesley, 2005



http://www.viewpoints-and-perspectives.info

Eoin Woods
Barclays Global Investors
eoin.woods@barclaysglobal.com
www.eoinwoods.info