Qualities and Constraints in IT Architecture

Non-Functional Requirements
Examples: Availability and Performance

Dr. Marcel Schlatter
IBM Distinguished Engineer
Member of the IBM Academy of Technology
marcel.schlatter@ch.ibm.com

IT Architecture Quality of Service Engineering — Speaker notes

These speaker notes are written largely for the benefit of the presenter, however in this presentation they also provide text which expands sometimes significantly on the content of the slides, and therefore there is no harm in also distributing these note to students if so desired.

The seminar is planned to run for 3 hours and contains 3 in-class interactive exercises. It is also accompanied by an optional after-class additional and larger exercise based around a case study scenario.

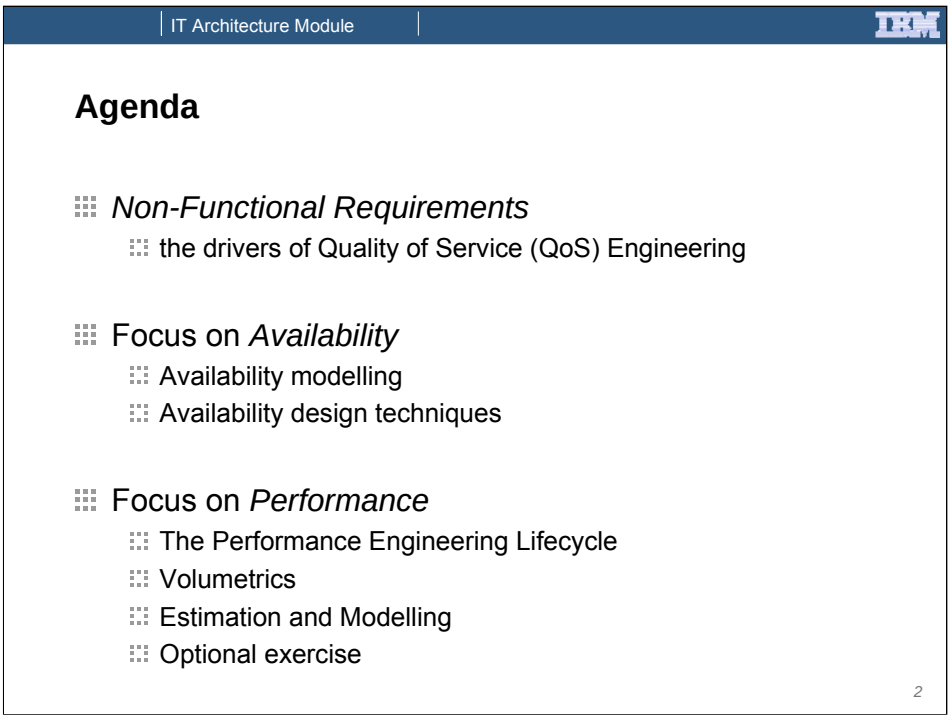
Session supporting materials

There are a number of files you should have been provided with along with this presentation file which support the materials and exercises contained within it:

1). A small simulation model demonstration in the form of a web page with an embedded Java applet. The files necessary for running this demo will have been provided as a zip file called 'simmodel.zip' which you should have unzipped to a directory somewhere on your presentation PC prior to the session.

2). The fourth Exercise in this lecture is based upon a small case Study called "AmGro-from-Home". This can optionally be offered to students as an assignment after the class. The case study introduction document for distribution to students is entitled "QoS Seminar - Exercise 4 input v*n.m*.doc". There is also an accompanying 'model' answer (not for distribution to students) in Microsoft Excel format entitled "QoS Seminar - Volumetric & Performance exercises full solution.xls"

———————-



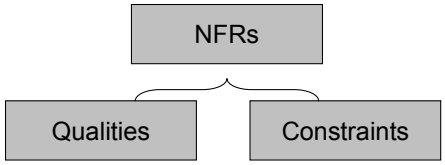
Agenda

The session starts out by charting the need for quality of service engineering in IT solutions through the consideration of the set of drivers we refer to as “non-functional requirements”. We present typical categories of non-functional requirements which should be collected and acted upon.

As there are so many different QoS disciplines and associated requirement types which can be considered for IT solutions, we have chosen to look at two of the most important ones – availability and performance – in some depth. This approach gives students some ‘hands-on’ experience of what it is like to performance QoS engineering activities in specific domains and in doing so will inform a wider appreciation of the subject.

Quality of Service Engineering starts with the '*Non-Functional Requirements*'

- Non-functional requirements (or NFRs) define the desirable qualities of a system and the constraints within which the system must be built
  - *Qualities* define the properties and characteristics which the delivered system should demonstrate
  - *Constraints* are the limitations, standards and environmental factors which must be taken into account in the solution



3

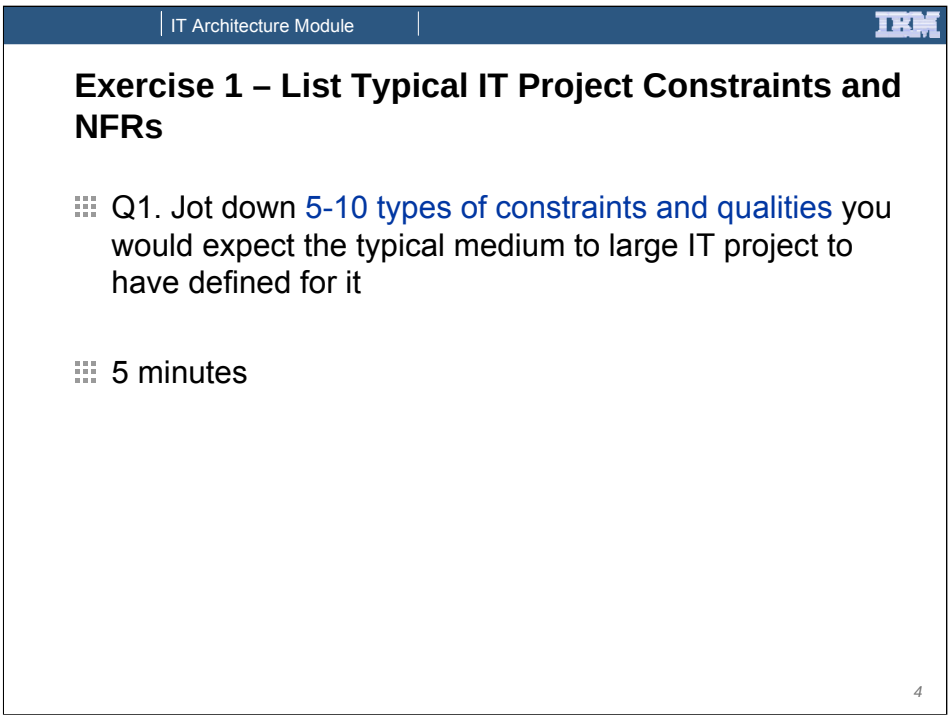
What are non-functional requirements?

Non-functional Requirements (or NFRs) describe the desirable qualities of the system under design and the constraints within which the system should be built. (These are distinct from the *functional requirements* which describe *what* the system should do in terms of processing inputs and giving outputs.)

Qualities define the expectations and characteristics that the system should support whereas **constraints** are limitations, prevailing truths, standards and any other relevant 'environmental' factors which must be taken into account in the design, build and running of the solution. Other lectures in this course cover the overall architectural method and subdisciplines – it should be noted that many of these disciplines take the non-functional requirements as their principle driver.

Note we can view qualities as 'forward' (positive) driving requirements – they describe wants or needs, desirable outcomes and properties of the future system; whereas we can view constraints as 'pushbacks', or perhaps 'negative' influences – holding us back (*constrain*ing us) and providing additional barriers or factors to deal with. The two categories of NFRs are intimately related of course - constraints may make it more difficult to achieve the desirable qualities, especially when they reduce our options.

E.g. the *qualities* of *this presentation* ought to be that it is clear, informative, interesting and helps the students achieve their academic goals. The *constraints* are that we only have 3 hours to do this in, and that (perhaps) the speaker is hung over, tired and has a cold (one of those usually applies to me in any case …)

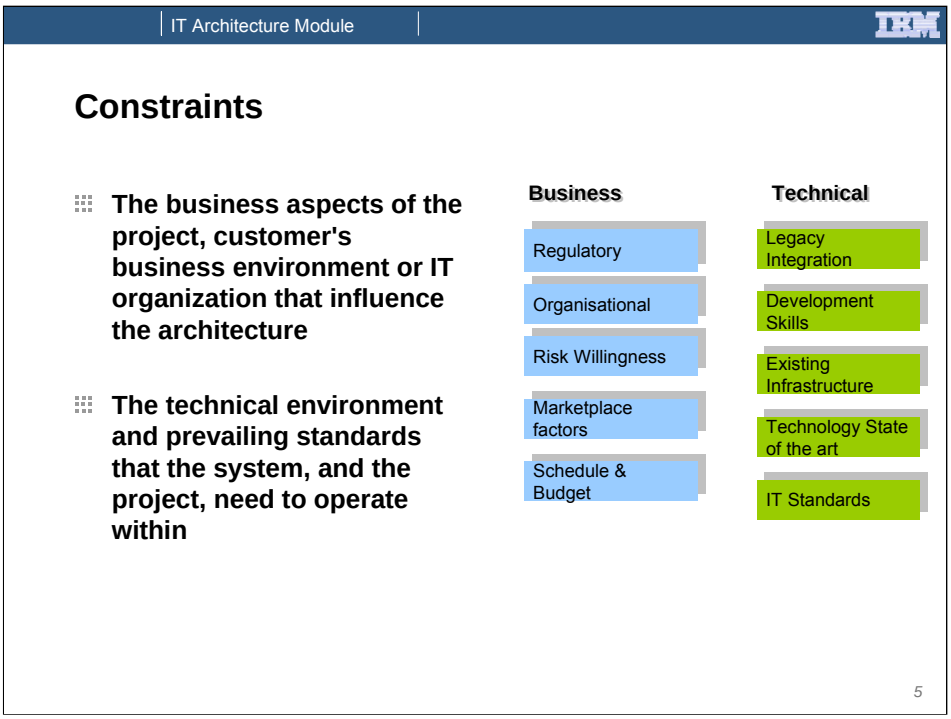


Exercise 1 – Students improvise types of NFRs they think are important

The purpose of this exercise is to start us thinking about the typical *types* or *categories* of NFRs IT projects should be subject to. Note that in real projects, key categories of requirements are often missed during requirements gathering!

It should be made clear to students that in this exercise, we do not want them to list *specific* requirements (e.g. “The system must respond in less than 2 seconds”), we are seeking *generic types* of requirement (e.g. “response times for online transactions”).

After 5 minutes, take a few minutes to collect suggestions of categories of requirements from the group at random and organise them (in one of two columns – qualities or constraints) on a whiteboard or flipchart – we will compare with them with the categories on the next two slides.



Types of Constraints

Constraints are all the factors, from whatever sources, that potentially limit or *constrain* what we may be able to achieve during the scope or lifetime of the project. On this slide we break these down into business-oriented and technology-oriented constraints.

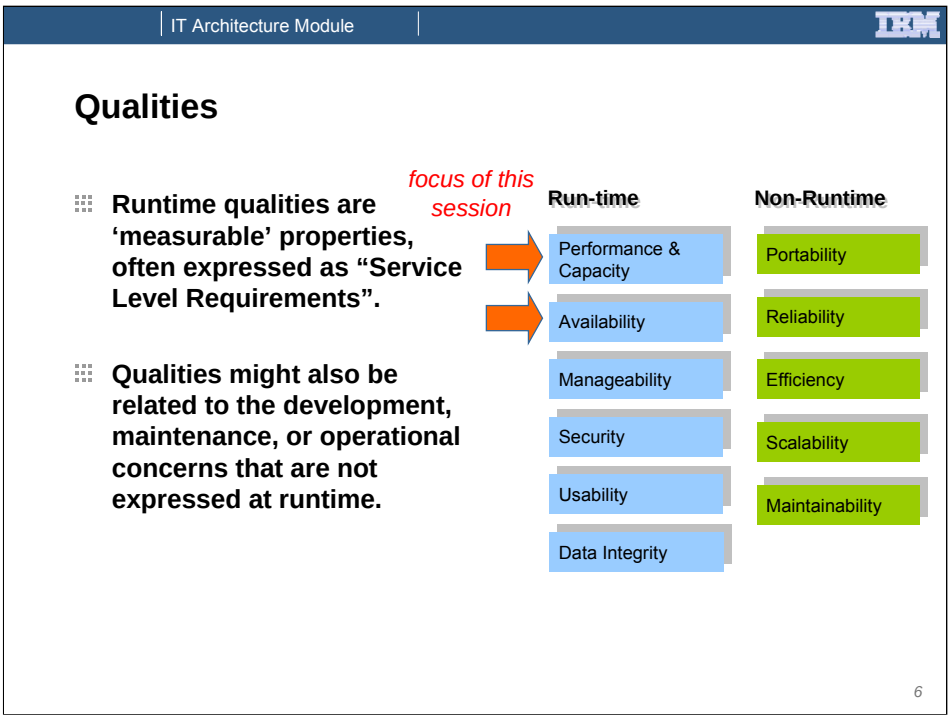
Constraints are often critical factors in making architecture decisions. For enough time and money, almost anything can be built, however the constraints help us define what is right or feasible for a specific client and project. Many projects that have failed have done so because they have failed to identify, understand and manage their critical constraints. So, careful consideration of constraints will help you deliver systems more effectively and successfully.

Technical constraints will include the existing IT infrastructure within the organisation, and the technical standards (often including specific products and technologies) which the customer will expect you to abide by. Often existing IT in a customer will be ageing, outdated and/or demonstrate poor quality of service (e.g. poor performance) already. Those existing applications and systems therefore not only constrain what technologies you can use for a new system, but also what overall levels of service you will be able to achieve when you integrate with those existing systems. Nevertheless, the customer may expect you as a matter of course to use the latest, greatest technologies in the implementation of your new solution, and will of course expect the new solution to be highly reliable and scalable – when you consider all of this alongside existing technology constraints, you may find the is a form of contradiction in the requirements – something will have to give.

Sp, before accepting constraints as true “givens”, they should be examined to ensure that they do not have:

- Outdated thinking, such as “This is the way we have always done things”, “It’s impossible to upgrade the network”
- Uninformed thinking, such as the "technology of the day"
- Biased thinking, such as "favourites” – personal and ‘political’ choices made based on what people want to learn, or who their friends are in the industry, rather than based on an objective view of what is good for business

Weigh constraints against qualities and requirements to clarify thinking. You should aim to understand you understand the true “strength” of every constraint – so you know what can be challenged and what things just have to be accepted as they are.



Qualities

Qualities, as already described, are the desirable, 'positive' properties that we want the delivered system to exhibit. We can subdivide these further - the qualities listed in blue on the slide are what are termed ***run-time* qualities**. Run-time qualities can be empirically tested (measured) by appropriately observing and monitoring the system either under test conditions or in live operation. Often prior to the running of a new system, the customer will agree with the operator of the system (which may be their own internal IT department or, in the case of outsourced IT, an external company providing that service) Service Level Requirements (so they become "Service Level Agreements" - SLAs) which specify how the system should perform in terms of run-time qualities. If the system does not meet these SLAs, their may be financial penalties which the service provider has to pay. This of course can act as one of the most potent incentives for doing good quality of service engineering during design and build phases!

Non-runtime qualities are more abstract properties of a system, such as its portability (ability to be moved do a different technology platform) or maintainability (how easy it is to update, change and maintain) and are therefore more difficult to objectively measure.

Another way of looking at NFRs is to consider whether they are ***quantitative*** or ***qualitative***. *Quantitative* NFRs express a requirement in numerical terms, such as "The user's logon request must be responded to in less then 15 seconds", "The system must 8,500 concurrently logged on users", "The service must be available for 10 hours each day, from 8.00am to 6.00pm". *Qualitative* requirements on the other hand are those expressed in subjective terms, e.g. "The system must be easy for new users to learn", "The user home page must be easy to navigate". Qualitative requirements can be more difficult to deal with as what constitutes success or failure is open to interpretation. Therefore, you should always attempt to establish a *success criteria* for any stated requirement to that you will be able to design a test for establishing whether the delivered system meets the requirement or not.

Qualities only become useful during architecture and design if you understand what aspect of a solution they affect – and bear in mind a single requirement such as a performance requirement will affect many aspects (hardware, software, runtime operations, …). It is also important to stress that Functional and Non-Functional Requirements must be related, i.e. Use cases and other functional statements must be associated with matching non-functional requirements, and reviewed for architectural viability. For example, every identified Business Event and Use Case should be associated with volumes and response time requirements (to support performance engineering); the criticality of each identified business service should be specified (to support availability engineering), the confidentiality of each data group should be captured (to support security design), and so on.

Quality of Service matters to business bottom lines

Tangible (a.k.a. our *quantitative* NFRs) are measurable and are the basis for the cost/benefit analysis. The impact of performance against measurable metrics can be estimated numerically, with the customers help or using "industry" cost metrics, to translate the impact of different qualities of service into business terms, e.g. :

• Slow sites and/or poor navigation techniques could be costing e-business companies over $100 million per month.

• In today's Internet world, the ratio of buyers to browsers on e-commerce sites is approximately 2%.....it's important to do a great job with that 2%.

• People leave ".com" sites because of pages being unavailable or downloads taking too long

• These tend to be quantifiable as a measure of “Loss per transaction”

Intangible metrics (a.k.a. our *qualitative* NFRs) are more difficult to perform a financial cost/benefit analysis on, however are still critical to consider:

• Any touch-point a business has with a customer, be it a website or a call centre, are projecting a company's brand image

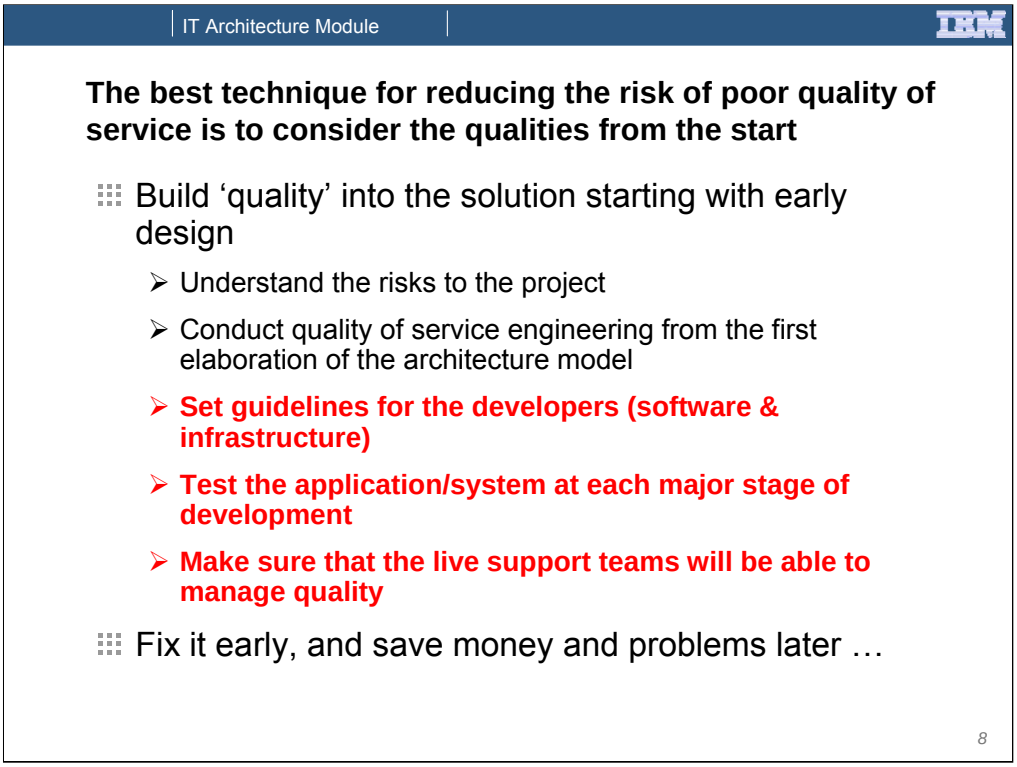
• Visiting a website is no different than going into a store with the company's logo on it .... a site experience affects the way people perceive the company

• A poor performing website (or one which is not there in the first place) is akin to getting bad service in a store

• Even if the experience produces no revenue, it can have an impact on whether a customer "returns to the store"

• The customer needs to develop a mechanism for calculating these “soft” costs

There are various techniques in the management consulting/management accounting domain for helping the customer work these costs and benefits out. The companies shown at the bottom are ones where IBM has had extensive involvement with the quality of service of the customer’s online brand.



System qualities such as performance should be considered from the start

There are alternative views as to how the 'quality of service' issue should be dealt with. Some organisations take a very 'suck-it-and-see' approach and perform very little quality of service engineering as a rule during system design and build – if things are bad in live then they'll attempt to fix the problems then. However, you can only take such an approach if you can afford the risk of a potential disaster in live – systems which run out of capacity for example will become unusable and hence effectively unavailable. Furthermore organisations taking this approach often find that their problems can easily be addressed in the live system due to inherent problems in the design or the technologies they have proceeded with.

So, the recommended approach is one of engineering quality into systems from the start – because the cost of repairing the problem, and risk to reputation will be greater the longer any problems are lefy unaddressed. Or put another way … if it takes $1 to fix a problem in design, it may take…

…$10 to fix the problem in build

…$100 to fix the problem during testing

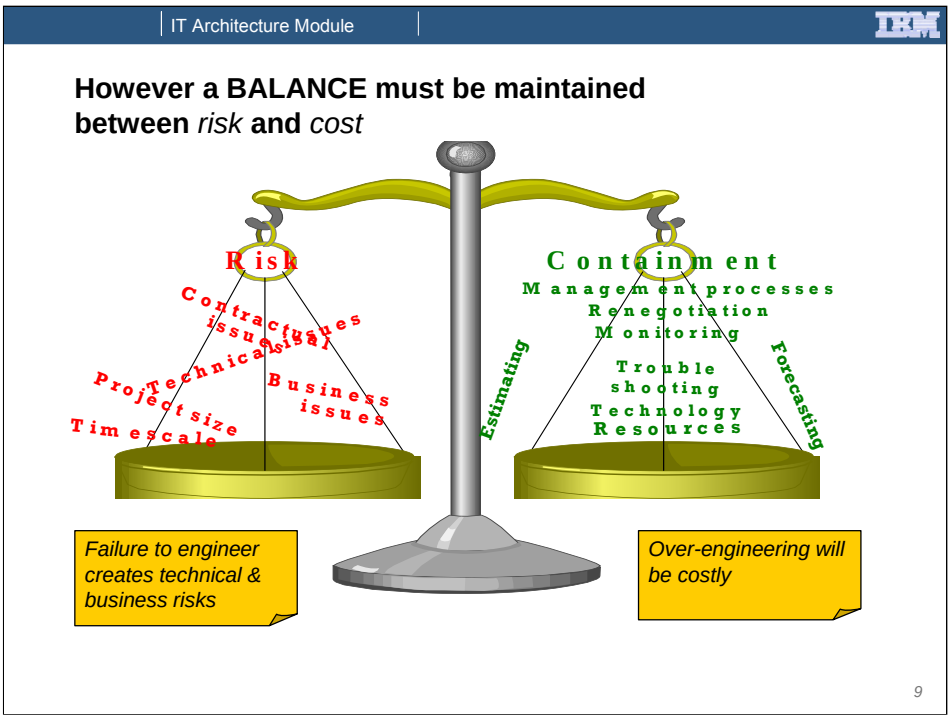
…$1000 to fix the problem during live operation

Early versions of the architectural models should be taking all of the non-functional requirements into account. It is important to build both the component (application) model and operational (infrastructure) model with requirements such as performance in mind. This is because properties such as performance are exigent from the complex interaction of the many layers of the system – not just isolated to hardware or software layers. Inherent application limitations such as single-threaded or single instance components fixing the problem could mean an expensive rewrite and redelivery of the system.

Therefore, developers and system builders should be provided with guidelines to follow. These guidelines should include both best practice approaches and wherever possible detailed targets and goals for them to meet. These should then be measured during the development/build phase or as part of unit testing in an attempt to discover and address any major problems at an early stage.

Such budgeting and unit testing activities are only one part of the overall testing which should be planned for a large project – non-functional quality tests should be planned for each major phase of the project lifecycle. Key in these are the large, scale load and technical tests of the integrated system (when all components and interfaces are brought together in a 'live-like' environment)

Taking into account the needs of those who will be supporting the live operation is also a important part of the overall approach. Care needs to be taken to ensure that systems have features built into them which make it possible for them to be able to successfully manage the availability, performance and security of the system in production. Always bear in mind that *real* system usage tends to vary from the "clean" and simplified scenarios simulated in testing, and that systems need to react to changing workloads (organic growth, new unforeseen users) and patterns of usage.

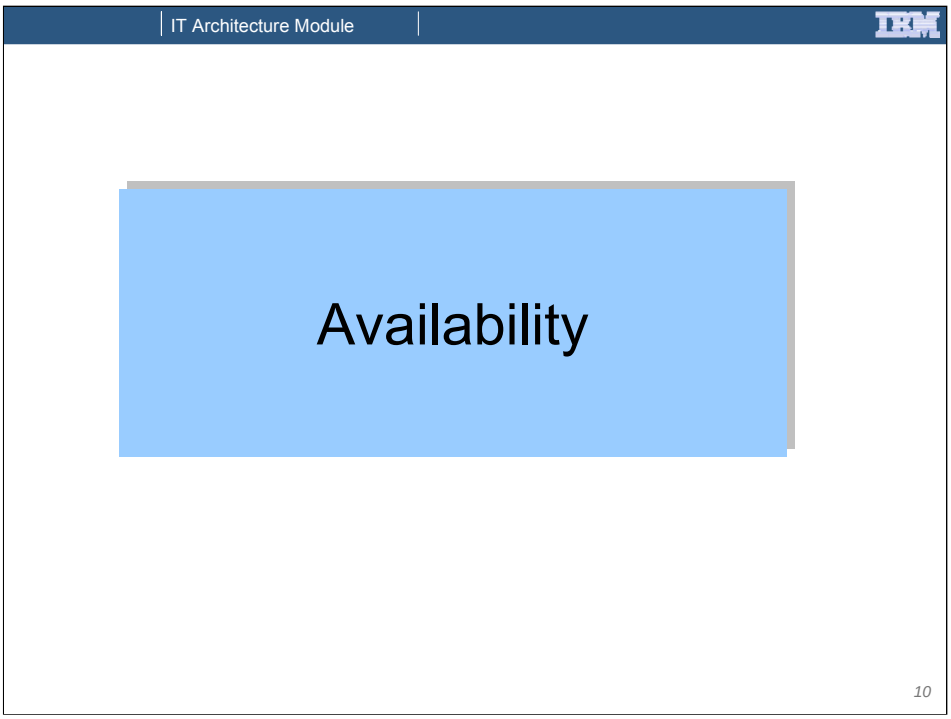


Requirements in the balance

Of course there is a balance to be struck as to amount of effort and time (= cost) to be put into your project plans for quality of service engineering activities. The better a system is engineered, the longer (generally) it will take to build and test and the more it will cost as a result.

That's why it is important to understand where the *main risks* are in the context of each project – i.e. where the impact of poor quality of service would have the greatest impact. Clearly the scale and criticality of the system to the business is a major influencer of how much effort is appropriate. Analysis of the key risks in light of this helps us focus effort in the most beneficial areas.

Let's also, seeing as I've got some space left on this page, take a moment to make an important note here about the word "requirement" (i.e. the 'R' in 'NFR') in this context. Whilst we might usually read the word requirement as meaning "something which must be provided" in reality things which are written into NFR documents are in fact "negotiables", and not in fact hard and fast requirements. The strength (forcefulness) of each documented "requirement" can vary greatly, from a "nice to have" to "absolutely must have". The extent to which this is acknowledged or not for each requirement depends on the maturity of the relationship and level of openness between the IT solution provider and the customer. We may not know which influences are the strong or the weak ones in a given situation, but in any case it our duty to point out which influences have the greatest impact in the inevitable trade-off between cost, quality and speed of delivery.



Availability

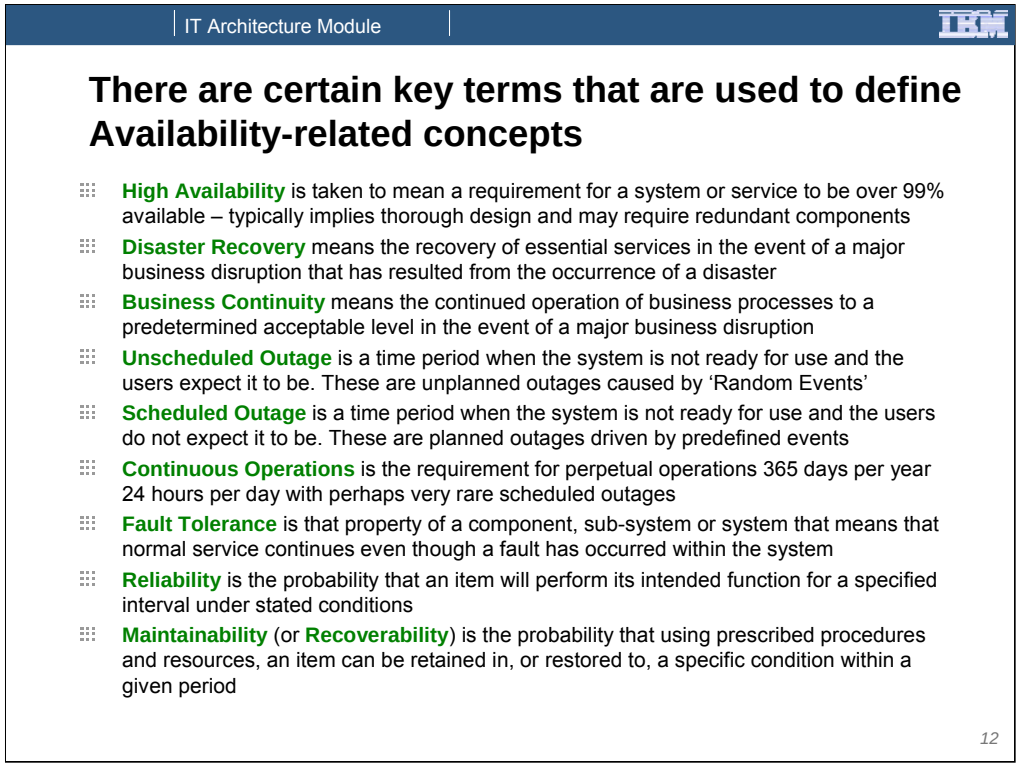
The reality of Availability is that customers directly relate it to the End User experience

The Availability of a system is a measure of its readiness for usage

Message: Availability is about end-user experience

Non-availability is the worst possible manifestation of poor quality of service – we can't use an unavailable service *whatsoever*. It doesn't really matter to the end user *why* a service is unavailable – just when it will be back online.

Note: in the picture, these women do at least have a fallback for the non-availability of their PCs (a pack of cards)! Some customers don't think about their fallback positions until it's too late!



Availability terms

These definitions are used in the "Designing for Availability" Technique Paper which is a technique contained within the IBM Global Services Method.

Customers in any given industry (e.g. banking) may prefer definitions which they feel more comfortable with. Developing these definitions with the customer up front is worthwhile in order to ensure a common level of understanding.

High Availability is not easy to define numerically but it is generally taken to mean a requirement for a service to be available more then 99% of the time. This level of availability needs thorough design and may require redundant components as well as good Systems Management, Problem & Change Management and Recovery Procedures.

Disaster Recovery means the recovery of essential services in the event of a major business disruption that has resulted from the occurrence of a disaster. This area encompasses the ability of the total design to transfer data and workload offsite and to restart the workload at a new site. The switch often involves a system and service outage but in the extreme the switch can be made completely seamless to important users.

Business Continuity means the continued operation of business processes to, at least, a predetermined acceptable level in the event of a major business disruption.

Continuous Operations is the requirement for perpetual operations 365 days per year 24 hours per day with perhaps very rare scheduled outages. This type of availability needs an extremely thorough design that anticipates many failure scenarios and masks these from the users through the use of very fast recovery or stand-in actions. This will almost certainly need functional as well as operational design – i.e. such a requirement cannot necessarily be solved by technology alone.

Unscheduled Outage is a time period when the system is not ready for use and the users expect it to be. These are unplanned outages. These are 'Random Events'.

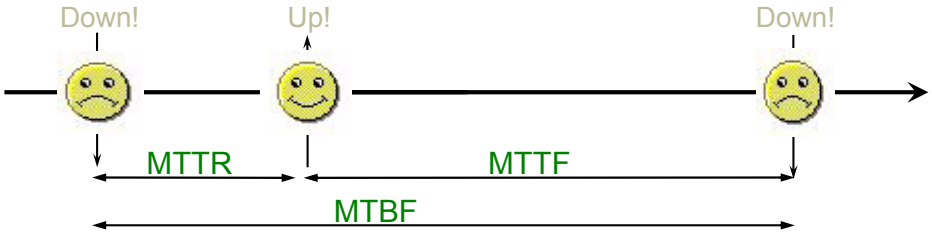
Scheduled Outage is a time period when the system is not ready for use and the users do not expect it to be. These are planned outages and therefore are predefined events, often at regular intervals (for example the first Sunday in each month).

Fault Tolerance is that property of a component, sub-system or system that means that normal service continues even though a fault has occurred within the system.

Reliability is the probability that an item will perform its intended function for a specified interval under stated conditions. Simply stated, it is how long the system can work. (Note: in a single component system used full time, Availability = Reliability)

Maintainability (or Recoverability). is the probability that if prescribed procedures and resources are used, an item will be retained in, or restored to, a specific condition within a given period

Key Availability terms – Mean Times …



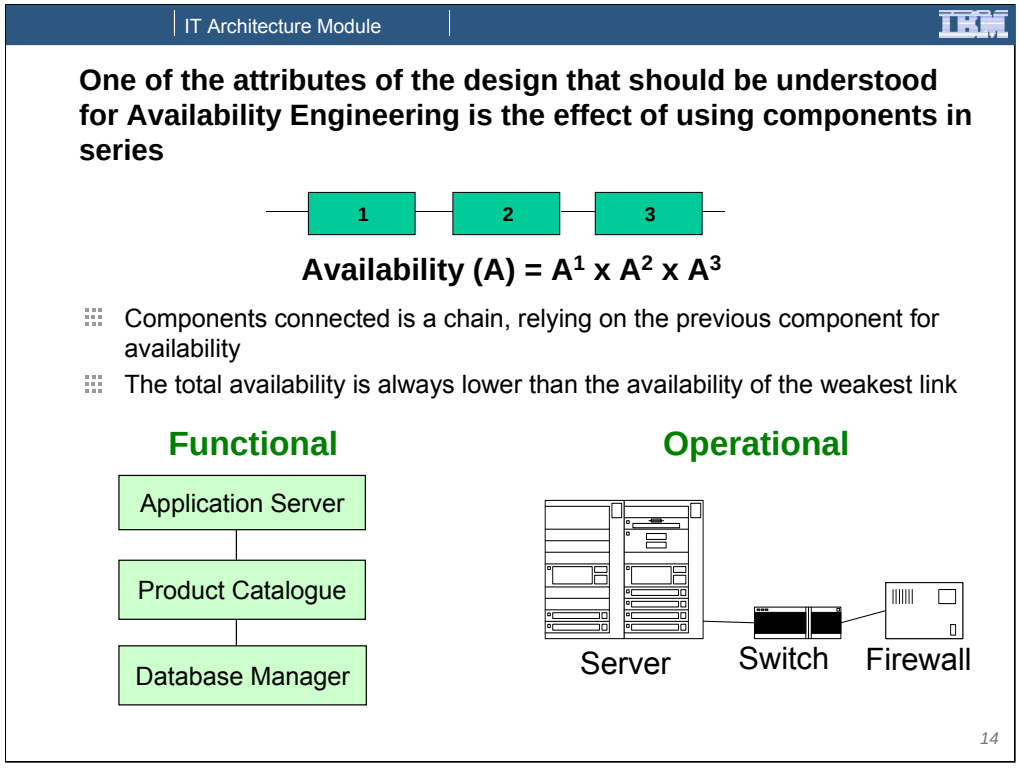
- **Mean Time to Recover** (**MTTR**) is the typical time that it takes to recover (includes repair) a component, sub-system or a system.
- **Mean Time to Failure** (**MTTF**) is the mean time between successive failures of a given component, sub-system or system.
- **Mean Time between Failure** (**MTBF**) is the average time between successive failures of a given component, sub-system or system

13

Mean Times …

Mean Time To Recover (MTTR) is the mean time that it takes to recover (and if necessary repair) a component, system or subsystem. Usually this is measured in seconds, minutes, or hours, but conceivably such values could be days or even months in the case of massive 'component' such as a data centre.

Mean Time To Failure (MTTF) is the mean time between successive failures of a given component or system from the point of uptime. Mean Time between Failure (MTBF) is effectively the same however also includes the recovery time (MTTR), i.e. MTTF is MTBF plus MTTR. But since MTBF and MTTF are usually much longer than MTTR (say months as opposed to hours) for most purposes the distinction is often unimportant.

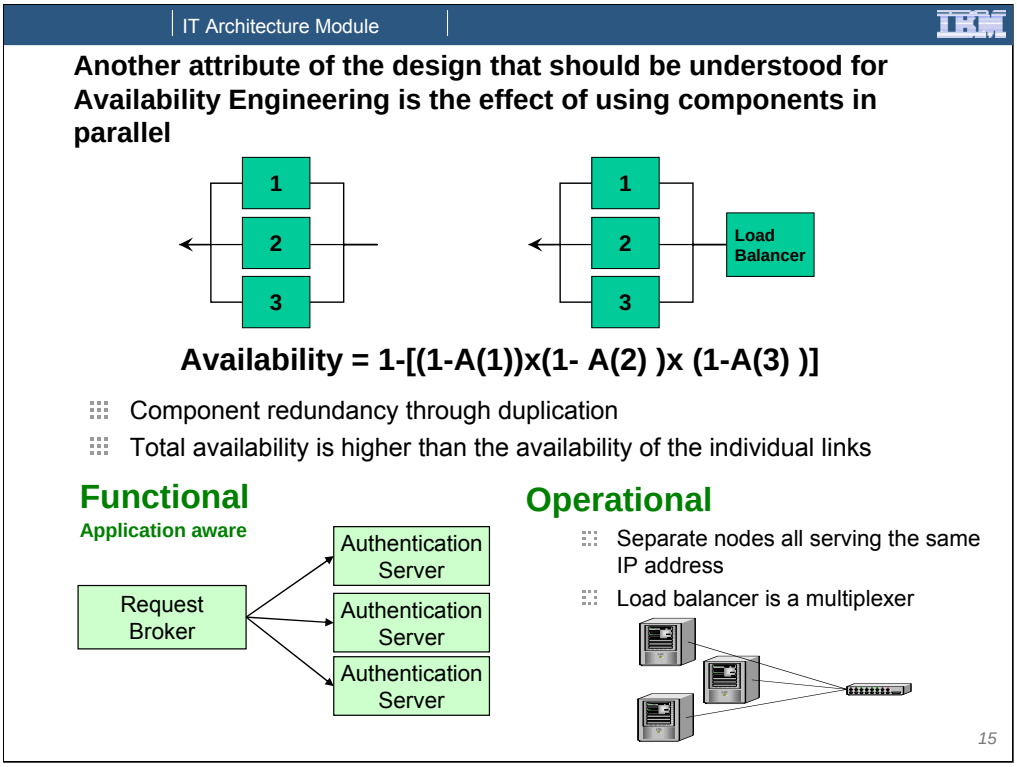


Modelling components in series

The diagram is an example of a "Reliability Block Diagram" (RBD).

Components in series rely on the whole chain to be there in order to provide the service. Therefore if any of the components fail the service they support will be unavailable.

Note we could ask the question: what level of detail should we model to? I.e. how many components are there in the serial chain? The level of detail to model to (as with any other QoS discipline) depends on the risk and how much trustworthy data you have available.

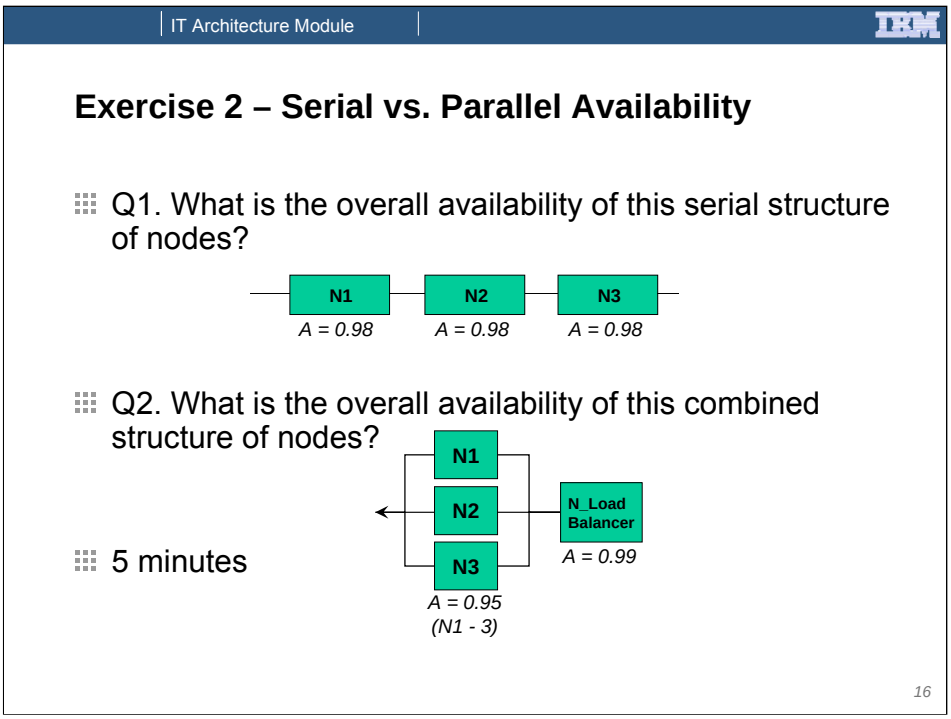


Modelling components in parallel

In the parallel model there are several routes through the system so long as the components in parallel execute equivalent function This provides us to consider a concept of ***alternate routing***. Messages, transactions, files being transferred, etc. can follow different paths through the system but still reach the same endpoint, and thereby the reliance on any one component is reduced or even eliminated.

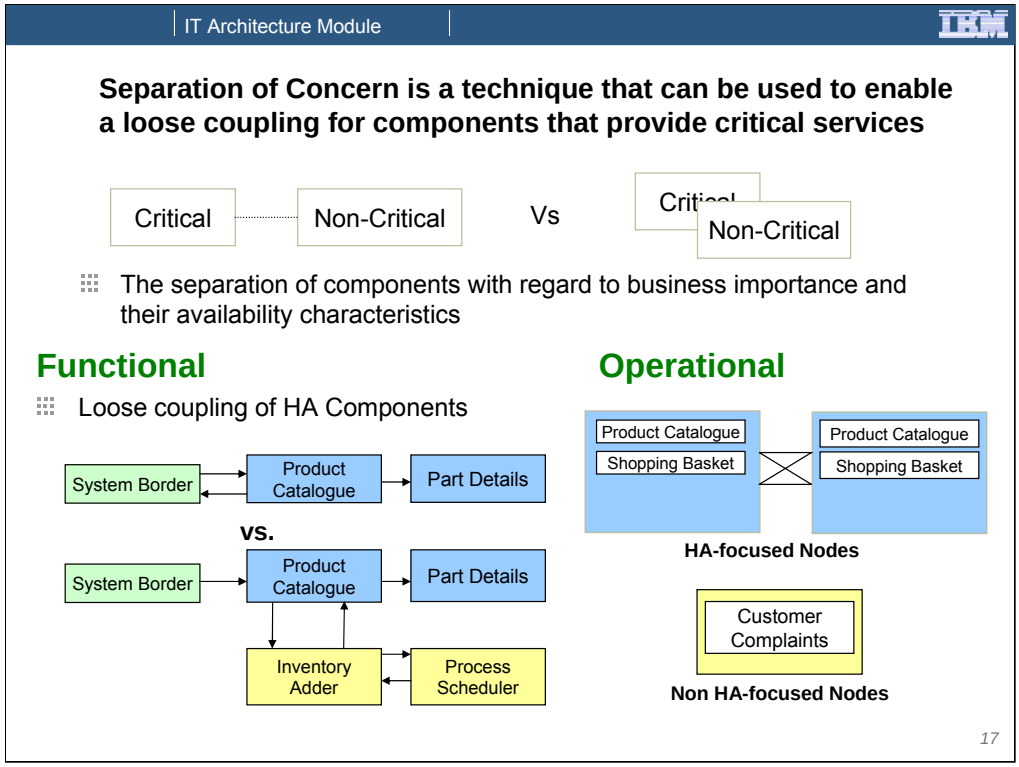
In systems you sometimes inherit alternate routing directly as a natural property of a component or you sometimes inherit it as an indirect property. One example of this would be to inherit it from a load balancing function. The prime reason for having a load balancing function is to distribute a workload across components so that throughput can be increased. In doing this however, alternate routes are provided.

Note that where a component (such as a Load Balancer) is providing the alternate routes, it may become an availability concern itself – it is required to operate or no routes will be available. This is because it has effectively become a component in a serial chain. To address this, parallel load balancers would be required (indeed this is a common configuration today).



Exercise 2 – Series vs. Parallel Availability

The known availability of each subcomponent is given by the side of each block (A = …) – the task is to calculate the availability of the total structure using the equations provided on the preceding slides.



Separation of concern

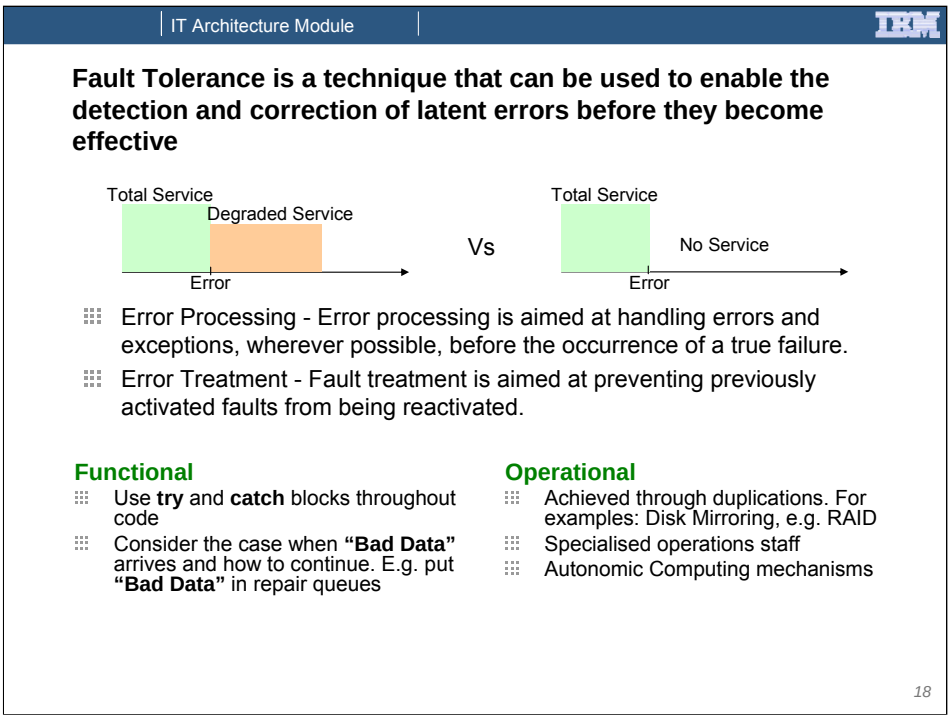
On the Operational side, this technique manifests itself as a matter of deployment. Once it is understood which of the key business services have to be made highly available then those key services should be deployed specifically across the Nodes where specific attention has been focused on High Availability.

On the Functional side, this manifests itself as ensuring that components that provide the HA required services are coupled correctly such that they do not depend on components that are not providing HA services.

Other related techniques in this space include:

• **Functional Isolation -** Through careful separation of business function (functional decomposition) it is possible to create application components within an application that although they rely on each other can still continue to function (perhaps with reduced function) even without a connection.

• **Technical Isolation** – the technical design can be configured so as to ensure that components to not put each other at risk through sharing of resources (connections, data, caches, middleware components, queues, etc.). This means instantiating resources separately, which is an overhead, but one which may be worthwhile in order to provide necessary degrees of protection.

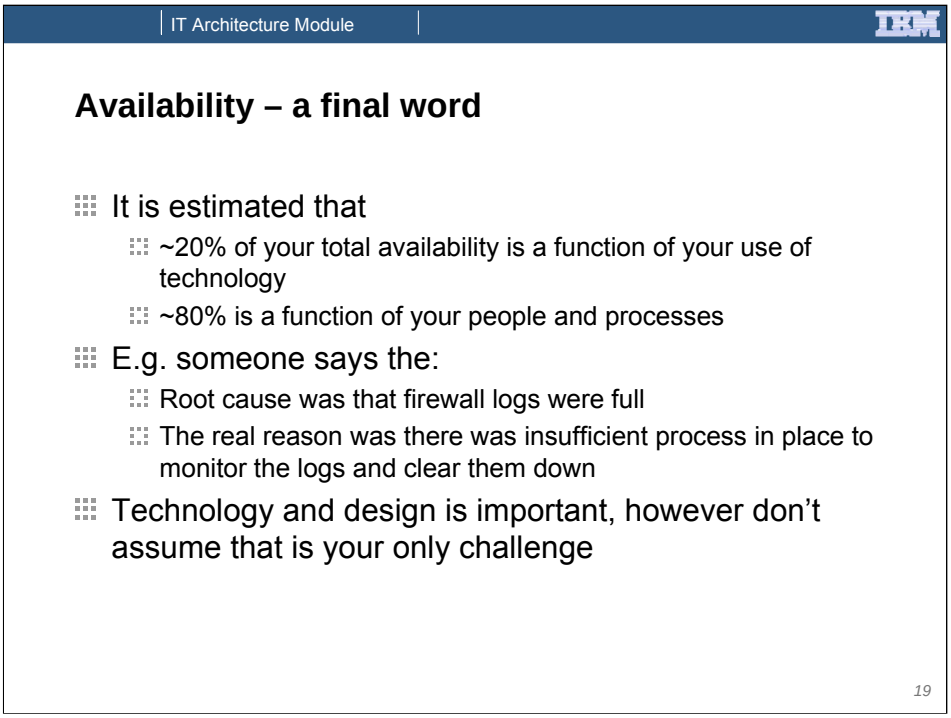


Fault tolerance

There are many different types of faults and exceptional conditions which our system components may encounter. The more comprehensively we design our system to gracefully and intelligently handle these possibilities, the more reliable and available our system will be.

On the left diagram, if we have an outage we still retain a degree of (degraded) service. Functionally we might achieve this by making sure we always catch and handle (as best we can) all possible exceptions within our applications. This is "Error Processing" – capturing errors and making them non-fatal to the system.

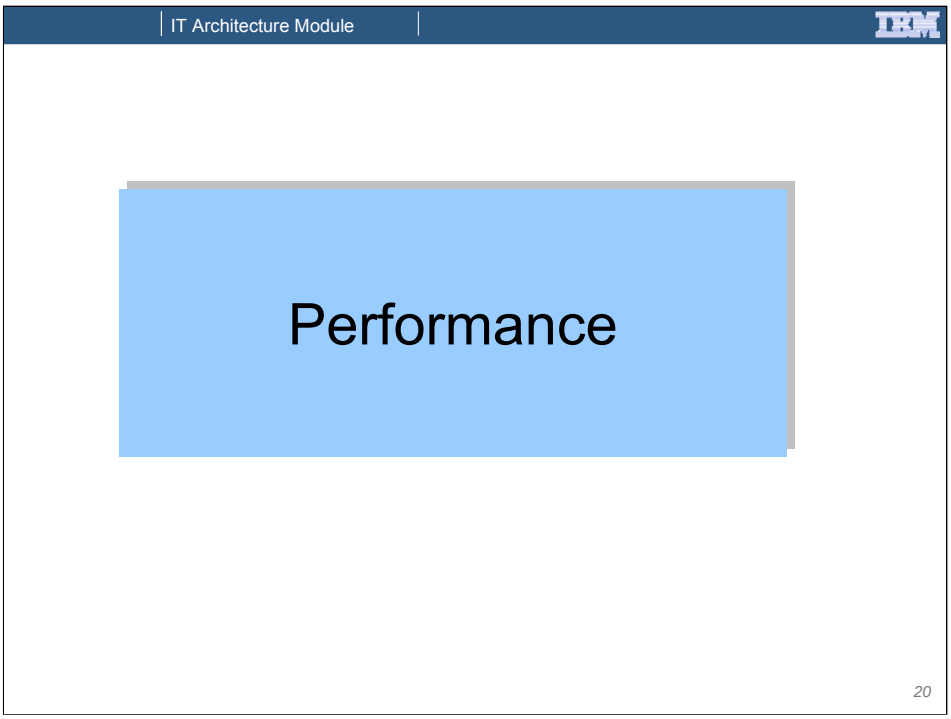
Error Treatment – having identified previous errors, this technique asks "What can we do to prevent their reoccurrence?" This could take many shapes – e.g. switch offending functions or components off until a proper fix available, or initiating a more full diagnosis of the root cause.



Availability – final words

This has been a relatively brief look at the subject of high availability – there are many aspects, angles and techniques which can be explored further.

We make the point on this slide however that *total* system/service availability is a function of both the technology solution *and* the people and processes which support systems in live. These two aspects of the total solution should of course be synergistic – the technology solution should support efficient and effective processes.



Performance

What is Performance?

- Definition
  - "Performance. The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage." [IEEE-610.12]
- In general
  - *Timeliness* of response, and *predictability*, are the two main goals
  - "Faster" is not always enough, as in for example, a real time system requires extremely consistent performance
- An (old) quote:
  - "A manager's goal should always be to strike the right balance between system function, processing costs, people costs, and performance. This is why the technical aspects of performance can never be entirely divorced from organizational politics"

Definitions in performance

In simple terms, we associate *performance* with speed – how quick something is. "High performance" means rapid response times and high throughput capabilities. A "high performance car" for example is one that can accelerate very fast (akin to response time) and achieve impressive top speeds (which is akin to throughput).

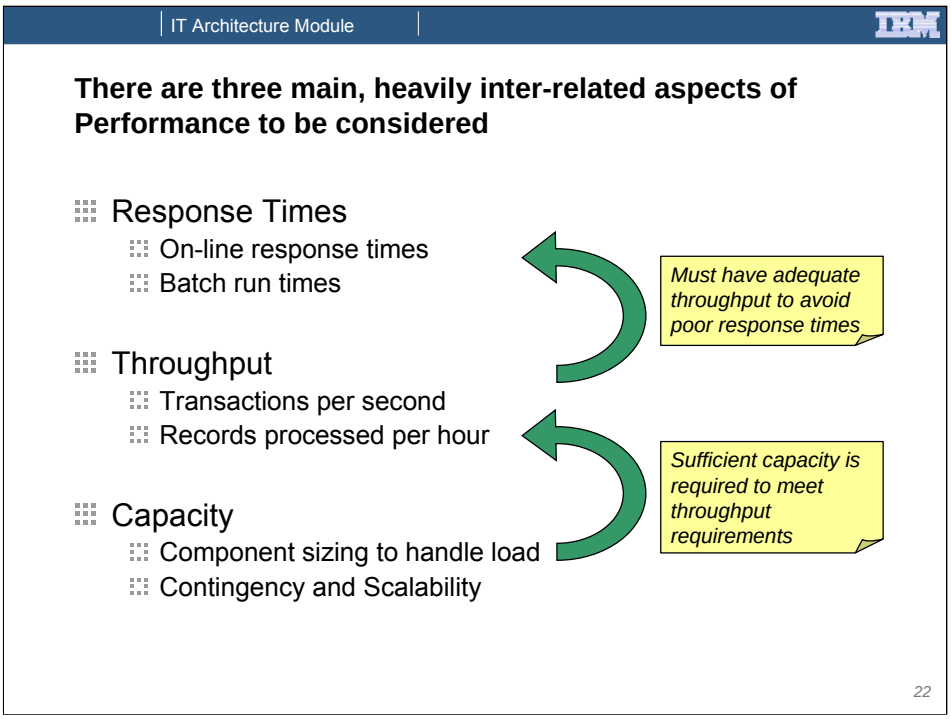
The definitions on the slide remind us that predictability, or constancy, is also a desirable feature of a system's performance. In real time applications (such as military systems) extremely consistent performance is demanded, but even in commercial IT, applications that have a consistent performance behaviour are much more acceptable to users than those whose performance varies greatly from one moment to the next.

When specifying response time requirements, we should make sure we understand the workload which the system will be expected to process whilst delivering that response time. A system may perform very acceptably under low load, but may begin to perform very poorly once the load in the system increases – ideally we will need to know what are the highest workloads the systems should be engineered to cope with. Only if we fully understand this will we be able to set and meet specific service level targets for the system once live.

Discussion points for the ICCM quote:

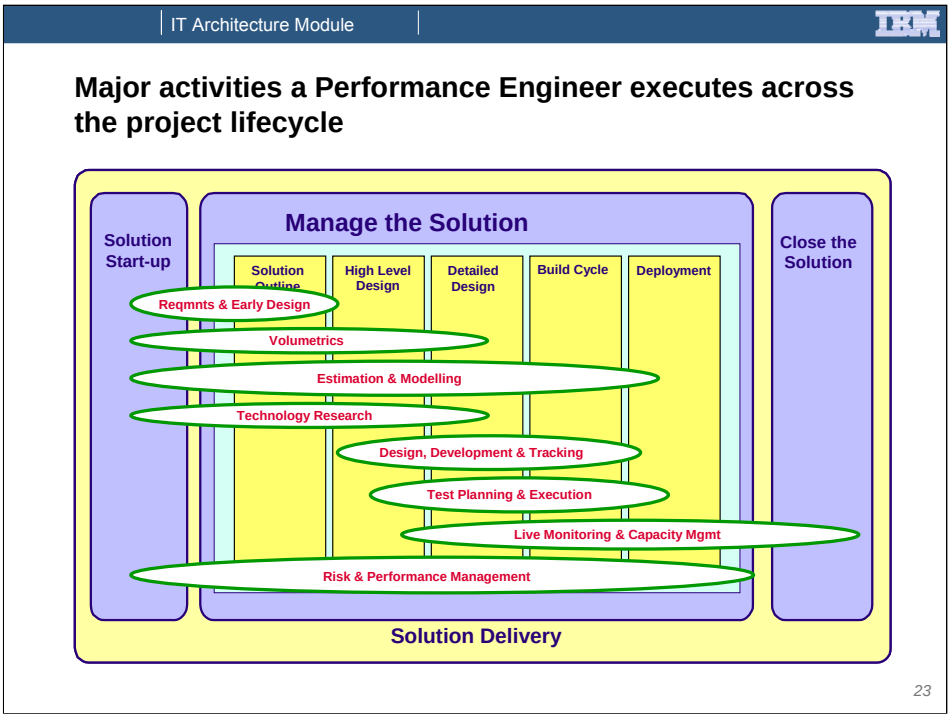
• Institute of Computer Capacity Management (ICCM), no longer in existence

• The statement talks about making sure that all aspects of performance are taken into account, not just the technical solutions - we could extend this to highlight the risk to the business of poor performance. This could well be the major factor in the justification of the effort we will require to deliver good performance.



The three main "pillars" of performance

These three terms and their interrelationships are fundamental to our understanding of performance. In one sentence, the performance challenge is to achieve satisfactory response times whilst supporting the necessary throughput without requiring excessive capacity.

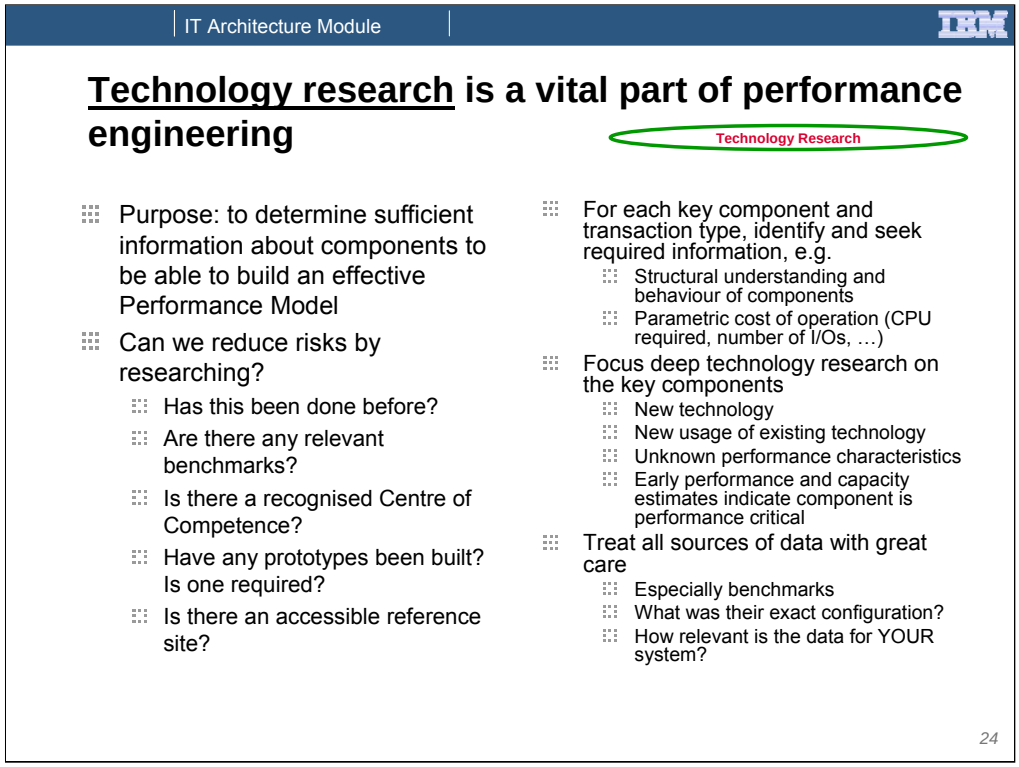


The Performance Engineering Lifecycle

This picture is taken from the IBM Performance Engineering and Management Method manual.

Regardless of the formal work products that any method will lead you to produce, there are a number of key threads, or groupings, of activity that need to be carried out across the phases of a project. These we will refer to as the performance engineering 'Themes' and they are shown in outline in the ellipses that overlay the solution phases diagram in the slide. We could relate the 'themes' to musical, architectural or decorative themes as the underlying flavour that gives a consistency to your combination of activities.

Keeping these conceptual threads in mind will help you to move the project steadily forward towards the achievement of your Performance Engineering objective – i.e. that of getting the performance of the solution 'right'. Each of these themes will be discussed in more detail later in this presentation material, however at this stage let us note they aim to be comprehensive encompassing everything from early requirements gathering to live monitoring and capacity management.



Technology Research theme

A key aim of technology research is to understand the proposed technology set and establish from this where the large risks might be. One of the main ways this can be done is by asking: "Is this solution similar to a previous implementation which can be used as a reference?"

If a particular technology has been used before, there may be existing benchmarks providing information about its performance (sizings, test results, etc.). **N.B.** We treat commercial benchmarks with a great deal of care – even scepticism – as a rule. Common commercial benchmarks and metrics, such as TPC-C and SpecInt can give an indication of the power of a system, but the transaction rates which are achieved in such benchmarks are typically heavily inflated. This is because the systems under test have been very heavily tuned by the vendors specifically for the benchmarks, and only the best results are published.

Other sources of information on technology can be very varied, and you should think laterally and be imaginative in searching for relevant data:

• Does anyone within your project have relevant experience?

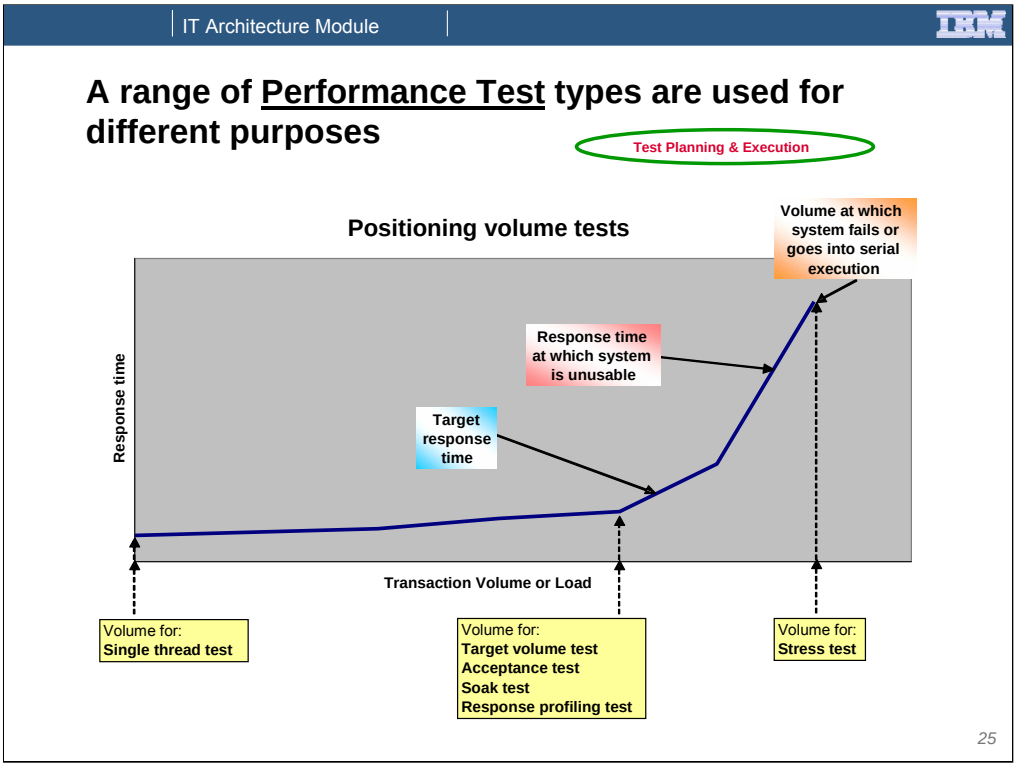
• Seek out vendor technical manuals and reference books (IBM Redbooks are a good example)

• Communities and Knowledge Networks within your own organisation

You will find that for most products and technologies, if you search hard enough you will find there are highly experienced individuals within organisations who are usually only too happy to share their in-depth experience and expertise with you. (N.B. They may have beards and sandals.)

Also bear in mind, many technologies can be tuned, tweaked and configured to improve performance in different situations. Part of your technology research should be to understand the different ways in which the technologies can be deployed and configured for different goals.

A small example – the Nagle algorithm: TCP/IP has a feature which stops networks being flooded with lots of small packets. These are stored until an acknowledgement has been received, and then transmitted. If, however, there is an infrequent request with a small size which must have reliable and speedy performance, this option can be disabled. (Real life example – LDAP requests in an intranet system). The option is TCP_NODELAY.



Performance Testing

The chart presents a lot of information overlaid together. Firstly, the underlying graph (blue line) depicts how a typical system behaves as transaction volumes increase – response times rise in a "knee curve" fashion above a certain workload. Secondly, we indicate (for an imagined system) the target response time and target load point. Finally, in the yellow coloured boxes, the types of tests which we may employ throughout the performance engineering lifecycle are positioned with respect to the load that they at are performed at. Note that the chart in fact shows a 'successful' case – where target volumes are reached before the response time begins to increase unacceptably.

With respect to types of tests, the system is tested at different levels of intensity, at different times of the project, for different reasons:

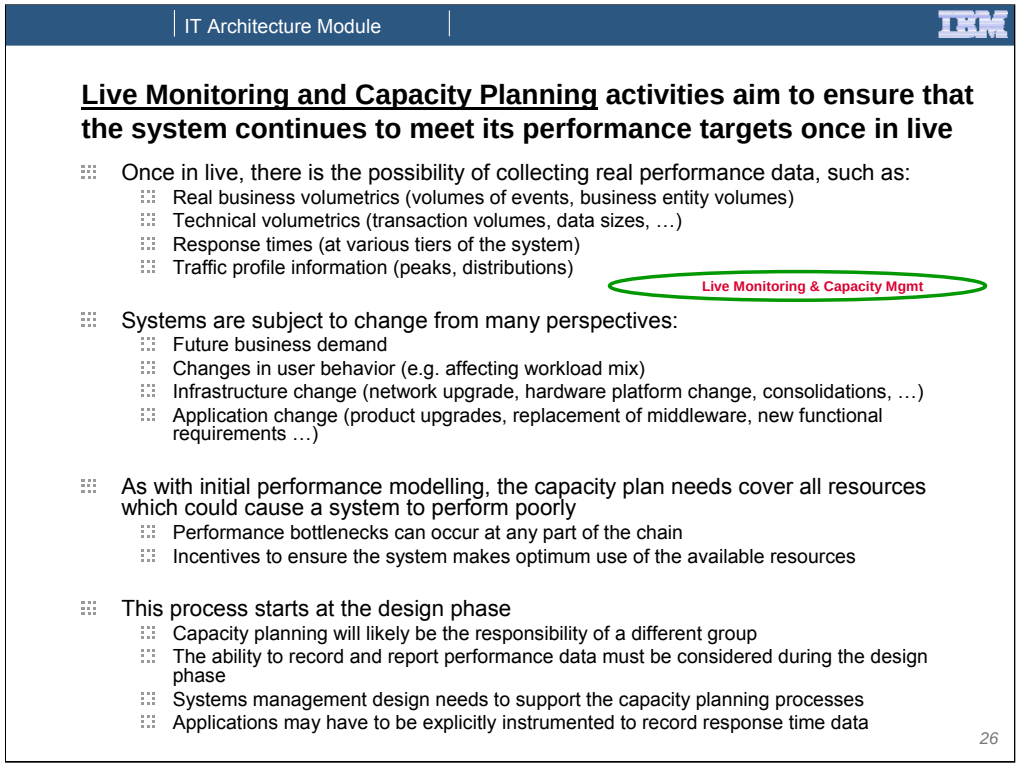
• **Single thread tests** typically take specific functions and execute them serially to understand the base response time with no load - if this isn't good enough, it's not going to get any better under load! This is a good way of tracking performance budget compliance during development (unit test time). Useful metrics for use in performance models can be gained from these tests.

• **Load/Volume tests**. Time permitting, volume tests should be run as many times, increasing this load gradually, so that the full performance profile of the system can be observed, and to discover at what points bottlenecks (which may be numerous) occur. For load tests, the workload mixture should be as realistic as possible / practicable – this need to be driven by the business volumes requirements

• The **Soak Test** is a special case, in that it runs as the full load for a long period of time (unlike the load tests, which may only focus on the peak hour) in order to identify any problems which may occur in live system operation. A good example of this is memory leaks in code – these usually only appear once a fair volume of traffic has passed the system, but the impact can be spectacular!

• The **Response Profiling test** looks more specifically at the way the response time of specific transactions change

• The **Stress Test** aims to break the system – it finds the point at which it stops becoming usable. This helps us understand how much headroom there is over and above the target load, to take account of unexpected peaks and future expansion.



Live Monitoring and Capacity Planning

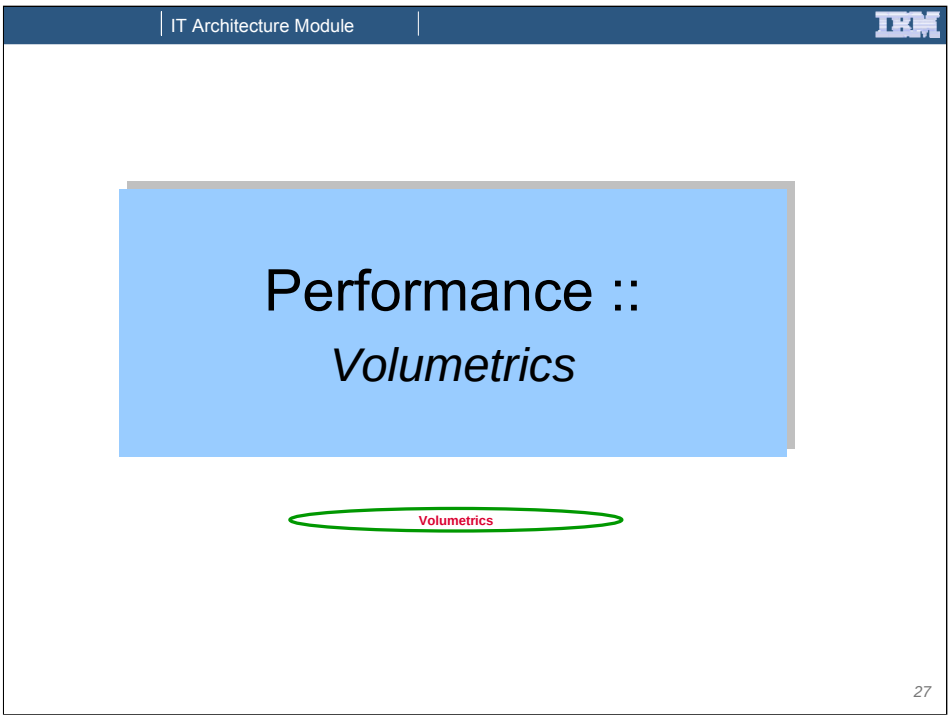
One of the key points from this slide is that during system *design and implementation* we must consider what mechanisms need to be implemented within the applications and infrastructure to allow successful monitoring and management of the system once live.

The mechanisms for capacity planning for future loads can be based on simple projections from standardly observable resource utilisation (cpu, memory, disk, etc) – such an approach is relatively straightforward and offered by many capacity planning tools, but it inevitably over-simplifies the picture. A fuller, more complete and more useful picture can be achieved if the application components of the system also provide data (response times, transaction counts, resource accesses, …) in support of capacity planning.

Whatever the mechanism of future systems capacity prediction, there should be an accurate model for prediction of future business demands and application of this to the capacity model. That way, the cost of meeting future business demands can be predicted.

The important thing is to take a complete, end-to-end view of capacity planning, as (for example) responses times can be affected by any 'link in the chain', from the user's workstation, to the networks over which they connect, to application servers and the back-end and external systems they access.

It is important to realise that capacity planning needs to be implemented in conjunction with the Service Delivery / Service Operator organisation. They will probably have existing Systems Management processes and standards within which the system being architected has to fit into. This should be considered throughout the lifecycle of the project. Conversely, you may actually have to challenge the Systems Management "standards" if it can be shown they will not be up to the task of managing the new system which you are implementing. Such step changes in capability can be difficult for Service Delivery organisations to justify so you may have your work cut out …



Volumetrics

Ok, we've whistled through a few of the Performance Engineering themes – now we will concentrate on one of the key ones, Volumetrics, in a little more detail.

The Importance of Numbers

Performance Architects rely on **VOLUMETRIC DATA** and **ASSUMPTIONS** in order to ….

What do you do when these are vague or difficult to get?

Feed **performance and capacity models**, in order to ….

Or difficult to map down to the technical level?

Predict system performance
- *online and batch*

Size systems
Evaluate & improve designs
Plan capacity
Plan testing

28

The Importance of Numbers

It's perhaps obvious to state that we rely on volume data in order to inform our analysis of likely system performance, but in fact the task of gathering sufficient and reliable volumetric data and assumptions often proves difficult for the reasons identified on the slide:

• Customer often find it difficult to actually predict or state their own future business volumes (in spite of the fact you would have thought they had a sound business case for doing what they are doing …)

• Seeking volumes in sensitive business areas can be subject to organisational and political barriers

• Even when business volumes are forthcoming, it can be difficult to translate, or map, these down to appropriate system level volumes. We will need a thorough understanding of the application and how it relates to business events and user activities in order to perform this mapping.

Enterprises often cannot provide detailed volumetric information – often, it has to be derived (or guessed!)

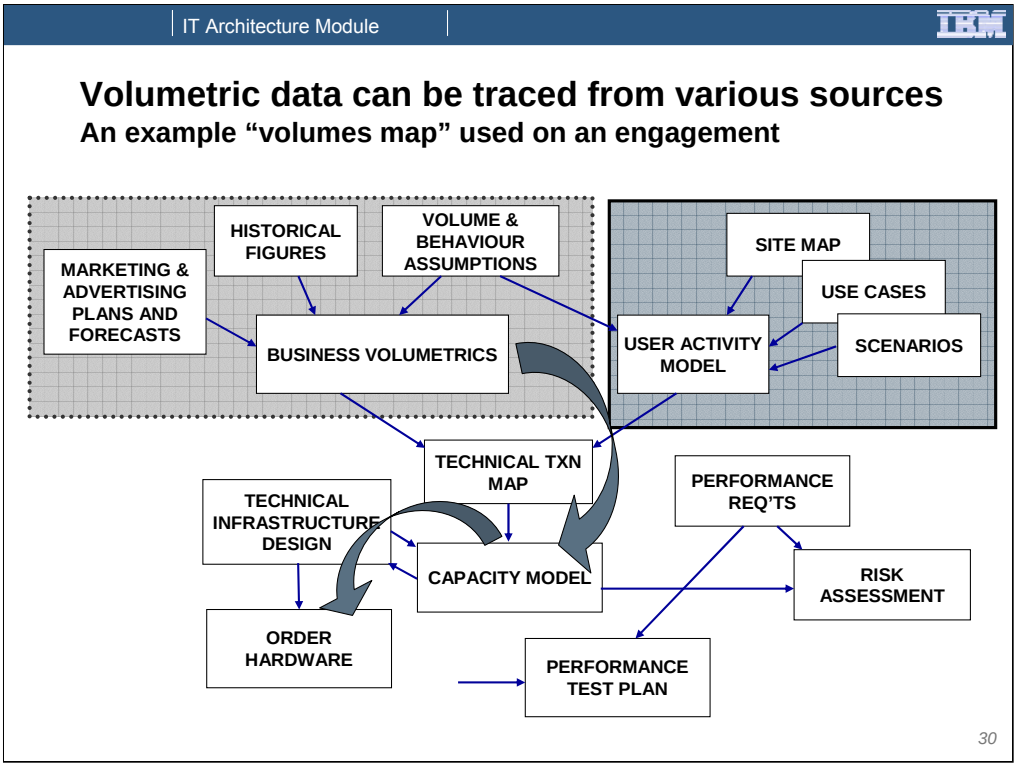
Real questions IBM Performance Engineers have been asked by customers

- *"We're just about to spend £20m on advertising our new brand. How many web servers do we need?"* - Insurance company
- *"Will this new digital audio broadcasting solution perform OK, given we don't know how we are going to use it yet?"* – Public service radio broadcaster
- *"How fast is the Internet?"* – Offshore bank

'Challenging' questions

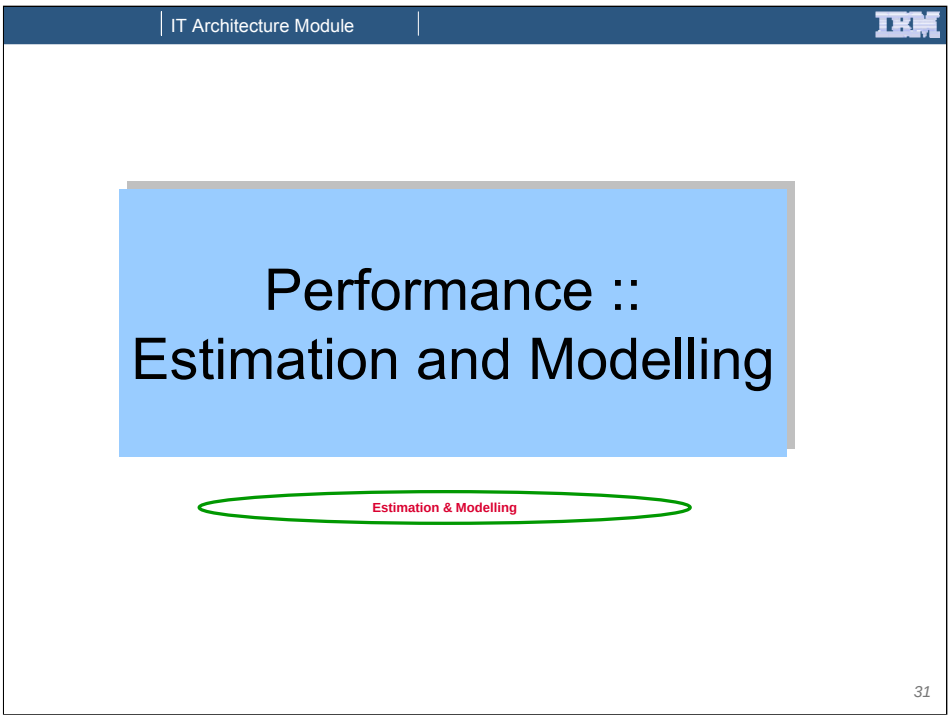
IBM Performance architects were asked these questions by real customers !! These real examples are used to demonstrate that customer often ask difficult, impossible or (dare I say) stupid questions!

In class, time permitting, students could suggest ways in which they would approach responding to these questions (take 5-10 minutes).



Volumetric data

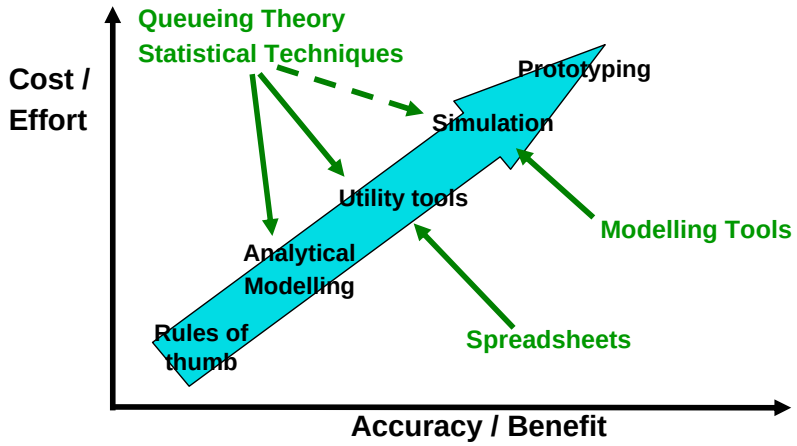
This data flow map was used in a customer engagement. Note how there are two paths to getting at the Technical Txn Map volumes – either via high level business volume assumptions or the User Activity Model. Whatever route is taken, the important thing is to be able to drive down to technical transaction volumes which can then be used within a performance and capacity model (and inform performance testing).



Performance Estimation and Modelling

Performance characteristics of a system can be investigated in more detail by creating a model

- Different techniques are available different levels of effort to provide answers with different levels of reliability



Performance modelling techniques

The key point from this slide is to establish the trade-off between effort and accuracy in performance prediction.

If the risk is low, simple estimation based on existing practice or experience may be sufficient to indicate that you can meet the objectives with ease. However, if the risk is higher and the performance target is more challenging you will have to put in more effort.

Rules of Thumb – a.k.a. folk tales: People who work with a particular solution all the time usually have an idea of some heuristic which can be applied. Solution vendors (say, if you are implementing a common package) will have rules of thumb also.

Hand calculation: Useful for 'quick and dirty' estimations of things like utilisation, response time for simple cases. Such calculations can be built up into complex cases, but takes a lot of data management. There may be, in this case, a utility tool to capture this.

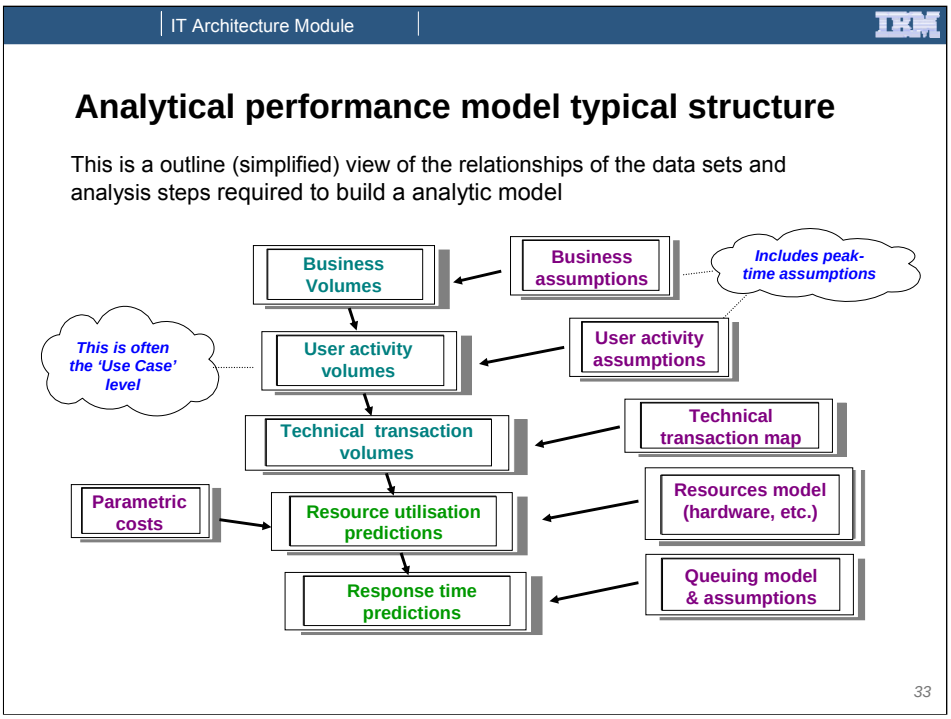
Analytical modelling / Utility tools - Effectively a way of organising your data in a meaningful way, and performing operations on it. 'Simplest', and most accessible is a model based on a spreadsheet Management can become complex, and predictions may be inaccurate depending on the complexity of the system you are modelling.

Simulation tools – simulation is a technique whereby the transaction, resources and interactions within a system are dynamically modelled. Some simulation tools will have limited domains in which they specialise (such as networks, web based systems) whereas others (e.g. HyPerformix tools) allow modelling of arbitrarily complex applications and infrastructure. Such tools do not use explicit queuing theory, rather they simulate the real systems which behave like queuing networks and the results are exigent from the simulation.

Prototyping – a technique whereby a prototype of the actual system is tested. Prototypes typically focus on specific areas and can produce accurate inputs for other models, which can then be used to perform 'what-if' analysis. Prototyping is important when it would otherwise be difficult to obtain useful assumptions to use for modelling, e.g. if performance atomics (path lengths for unit operations) are not be known.

You could treat Testing as an additional way of understanding the system. Testing results are extremely useful for validating models, both in terms of atomic assumptions and overall predictions.

Garbage in, garbage out! Always remember that, regardless of the chosen technique, the input data is often the most inaccurate part of the estimation process. If you are going into a more refined estimation process you also need to refine the input data until you have at least the same confidence level in your input data as you want from your estimate.

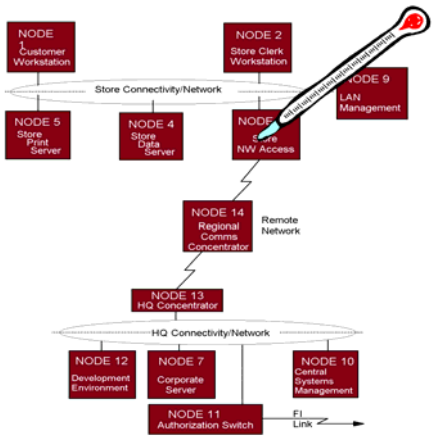


Analytical performance models

Note how volumes (taken from business to technical) are required to drive out utilisation predictions, and from there response time predictions are possible as we move down this chain. On the right and left around the central 'column' are all the input pieces of data which are required to build an accurate model.

The 'hotspot' concept is a good way of understanding where to conduct detailed performance analysis

- Typical questions you should seek the answers to …
  - Where are important functions (s/w execution) and data going to be located?
  - What rate of transactions will be required?
  - Are there any large volumes of data being sent over low bandwidth links?
  - What is the impact of multiple instances?
    - e.g. many branches to one data-centre
  - ➢ *Which parts of the architecture may prove to be 'bottlenecks'?*



34

Hotspots

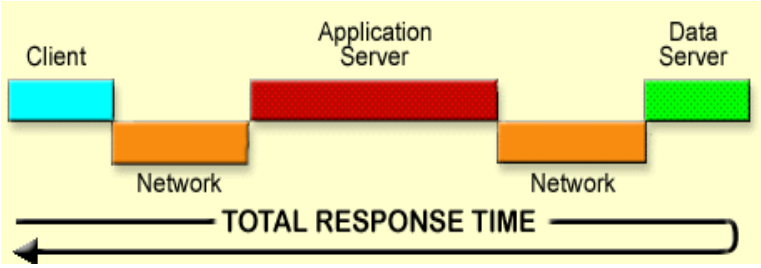
Regardless of your modelling technique, one of the major purposes of the exercise is to discover bottlenecks – the points in the system which will constrain performance. This is not a formal technique within the method, however, it is a useful way of thinking about the system as a whole. Experienced Performance Architects tend to develop a good 'nose' for sensing out where bottlenecks are likely to arise and can often jump quickly to the key parts of system either during development or when problems have arisen in live.

Starting with a design view of the system, such as the Component Model or Operational Model, think about how interactions travel around the architecture, including both online and batch data transfers (e.g. file extract and transfer, replication, backup, synchronisation). Analysing just the application model can be rewarding in itself – helping to uncover logical bottlenecks such as, for example, single instances of components, or serialised queuing mechanisms. Also bear in mind that in some projects the application model will be developed much earlier than the infrastructure model, and fixing problems here will make building a successful infrastructure much easier.

The Operational (infrastructure) model can be analysed for physical bottlenecks – e.g. all traffic to a single central server travels down a shared connection of limited bandwidth. Physical bottlenecks can be anything in the end-to-end path of transactions – network connections, network components, application servers, database servers, external systems interfaces.

We need to sum response times for all components an end-to-end transaction relies on

- Utilisation of each resource based on the total workload and workmix
- End-to-end response times based on multiple steps in the end-to-end transaction path

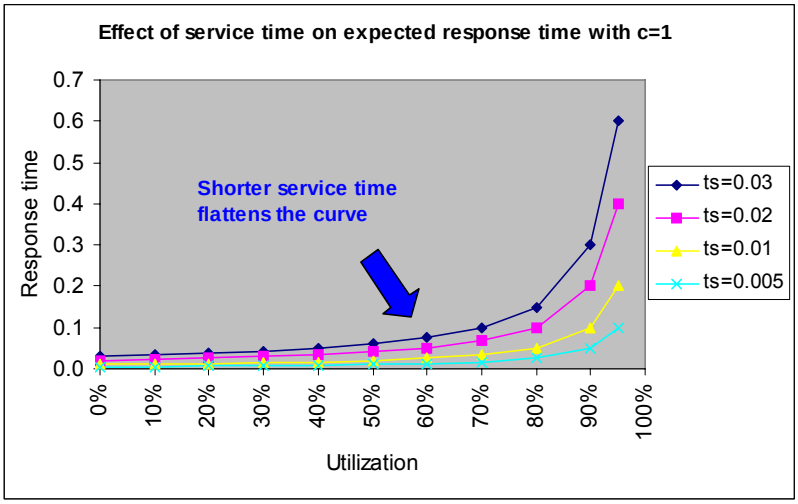


35

Response times end-to-end

The example is simplistic – the Application Server tier itself may break down into many parts. Furthermore a single ‘user-level’ transaction may lead to many technical transactions which operate serially and/or in parallel.

Response times degrade severely at high utilisations



Response time vs. utilisation curve

A 'server' in both this and the previous slide need not be a piece of hardware; it could also be:

A single threaded software process serving a queue of requests

A logical resource such as a database lock

A network resource, such as a shared line

This chart shows that as the utilisation of any resource increases, the overall response time increases. This is because of a longer time spent *waiting* (queuing) for the server at high utilisations, not because the processing in the server is any slower. As a rule of thumb: for a single server, if utilisation begins to exceed 70%, then response times may start to increase sharply.

This is intuitive, if compared with the very human example of waiting at an cashpoint … As the server (i.e. cashpoint machine) utilisation increases (i.e. more people in a given time period use the ATM), the average time that you will need to wait will increase. Of course, during periods of high utilisation, it would still be possible by chance to experience a fast response, if you happened to arrive at one of the few periods when the 'server' was not actually busy.

So it is logical to conclude that overall response time for access to a server can be improved by:

• Reducing 'server' utilisation (e.g. by scaling horizontally by adding more 'servers')

• Reducing service time (i.e. using a faster 'server' or reducing path length)

In terms of relating this to specific modelling techniques, in analytical models we need to include formulae to account for queuing factors if attempting response time predictions. In simulation models or in the measurement of real systems, the queuing effects will arise naturally.

Simple queuing formulae take no account of different priorities of workloads. In sophisticated operating systems (e.g. for many years in mainframe operating system) highly effective priority mechanisms are built into the o/s scheduling algorithms so that good response can be retained for high priority workloads even though the overall system may be very highly (or even 100%) utilised.

Exercise 3 - Volumetric estimation

- Shop
  - In the peak hour, on the average, every 60 seconds a new shopper arrives (random arrivals, generated by a Poisson process)
  - Average shopping time: 10 minutes (random distribution)
  - Average time at the cashier: 2 minutes (random distribution)

- Estimate the minimum number of carts the shop must have to make sure that customers almost never have to wait for a cart
- Estimate the minimum number of cashiers required to make sure that the number of customers that must wait for a cashier is almost always at most 3

The demo uses the Ptolemy II simulation modelling tool

Open Source simulation toolkit written in Java available from http://ptolemy.eecs.berkeley.edu/ptolemyII

The model is a Discrete Event simulator. It has been extended with some custom actors (in porkbench.jar)

37

25 Carts and 5 Cashiers are roughly sufficient; 6 Cashiers is safe.

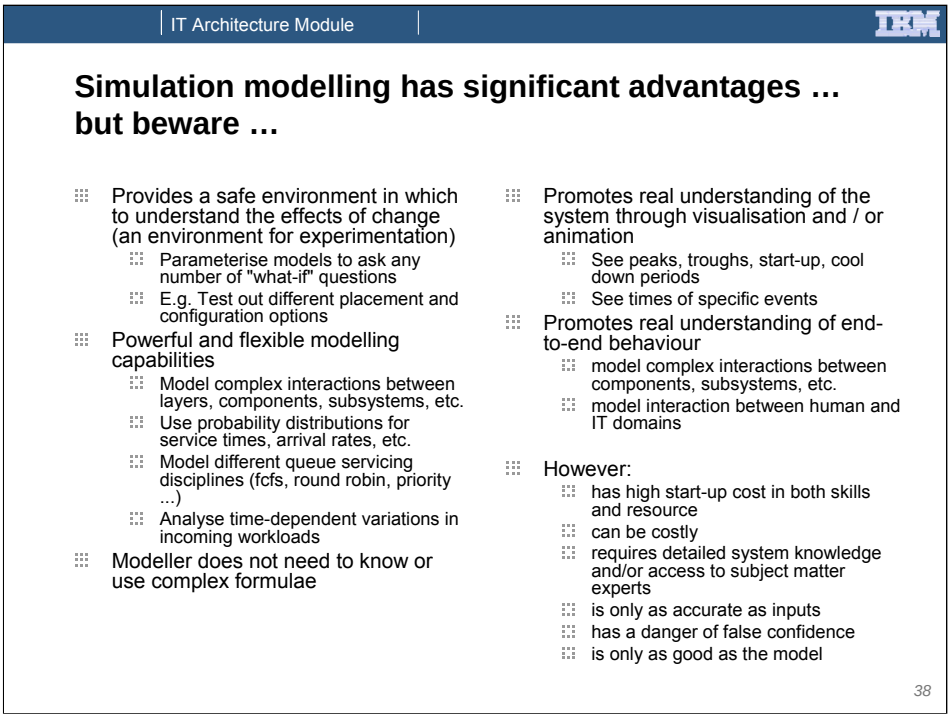
The graphs are:

Queue Length for Carts and Tills. Realtime display of number of shoppers waiting for the respective resources.

Resources - carts. How many carts are left in the trolley park

Utilisation - average and realtime utilisation of the tills. You can never use half a till at any given point in time, so (for example) a utilisation of 75% when there are only two shop assistants is impossible. The utilisation, which we are used to dealing, is an average over a period of time.

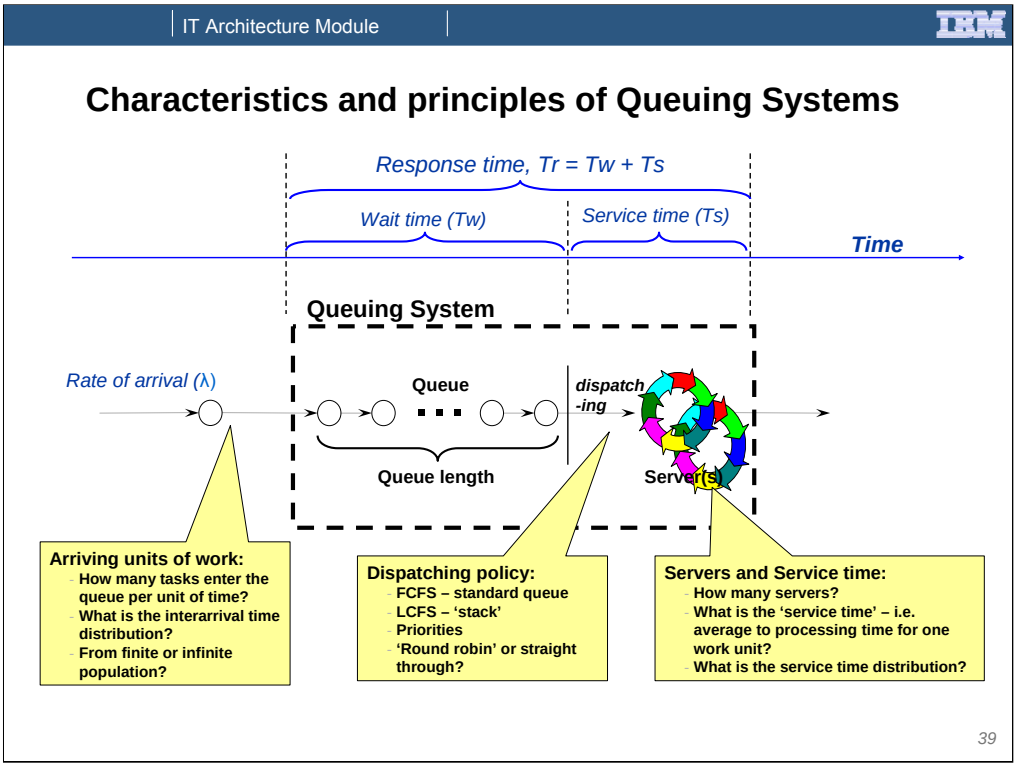
Time to complete shop - this is a combination of a couple of exponential distributions and some queueing. Not surprisingly it's a bit busy. The average is ot easy to see, unfortunately



Simulation modelling advantages and disadvantages

Like the other performance modelling techniques, simulation modelling has its strengths and weaknesses. In terms of a *design-time* approach (i.e. as a technique to use before any code has been written, any hardware purchased or anything installed), it remains the most powerful technique for predicting system performance, but this power comes at a cost. Professional simulation modelling tools are typically very expensive (tens of thousands of $/K and upwards for single licenses) and to extract the maximum value, significant time and effort has to be put into model construction by relatively highly skilled individuals.

A side effect of simulation modelling exercises however is that the process of building the model often causes questions to be asked which otherwise would not be (as the modeller needs to pry into system components and their dynamic behaviour). The impact of this process can be just as illuminating as the model results themselves …



Queueing systems

The schematic in the slide demonstrates the features of any queue for service. It should be emphasised that for accurate queueing analysis, it is important to understand not just averages but the *distributions* of key parameters such as interarrival times and service times. If interarrival times are highly regular, then the workload is generally easier to deal with, whereas if the input workloads are very 'bursty' (highly variable interarrival times) then it is more challenging to build a system which can guarantee to meet a particular performance level due to the queueing effects. The distribution for user-driven IT system input workloads is often assumed to be based on the *Poisson distribution*, as this is the classic statistical distribution used for representing inputs where events occur with a known average rate, and are independent of the time since the last event.

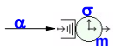
Note that queuing systems arise in many walks of life. Common on-the-streets examples include queues in shops and supermarkets, queues at cashpoint machines, and call queues in Call Centres. Regardless of the scale or nature of queues, the mathematics around them remains essentially the same.

Note that the *population* of a queueing system is defined as the number of requests (transactions) either queueing or being processed at a given time.

Notation for major types of queues

Formal notation for queues: α/σ/m, where:

- α Type of probability distribution that represents the periods between arrivals into the queue (M = Exponential, D = Deterministic, G = General)
- σ Type of probability distribution that represents the periods required to service each request in the queue (values as above)
- m Number of servers at the queuing center



Note: Other factors can be specified which define more advanced queue types, including:

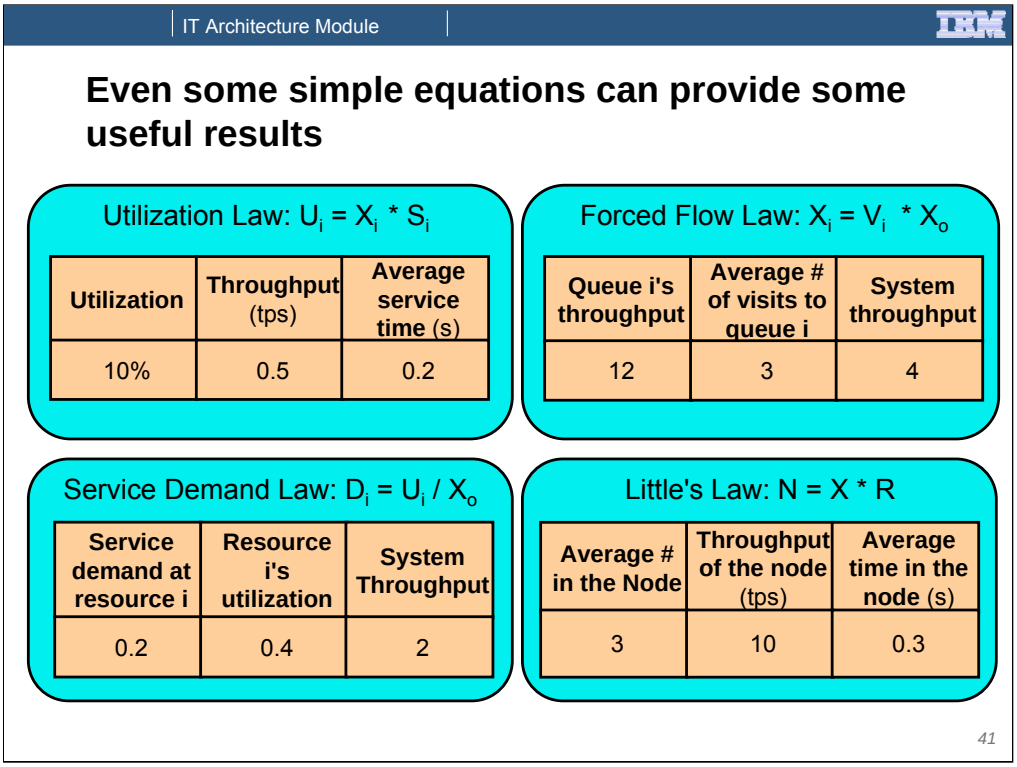
- Buffer Size or storage capacity in the queue
- The allowed population size, which may be finite or infinite
- The type of service policy e.g. FIFO, LIFO, RR, PS

M/M/1	M/M/k	Others, e.g. M/G/k
• Known as the 'Poisson process' • Exponentially distributed interarrival and service times around known averages; single server • Reasonable approximation to most single server queues • Mathematics are manageable	• Exponentially distributed interarrival and service times around known averages • Multiple servers • Mathematics more complicated	• Generalised distributions of either interarrival and service times • Multiple servers • Mathematics beyond most of us • If important, consider specialist tooling or simulation

40

Notation for major queue types

It should be noted that the mathematics of anything but simple queues rapidly become challenging to most non-mathematicians. In fact, whole degree level modules can be and are taught on queuing theory alone – it is not our intention to give you such a course – however it is useful for IT Architects to be aware of the different types of queues which can be modelled, and the limitations of standard formulae.



Simple laws

These simple laws relate average / macro level properties of systems such as throughput, average service time and utilisation. An example using these laws follows right on the next page…

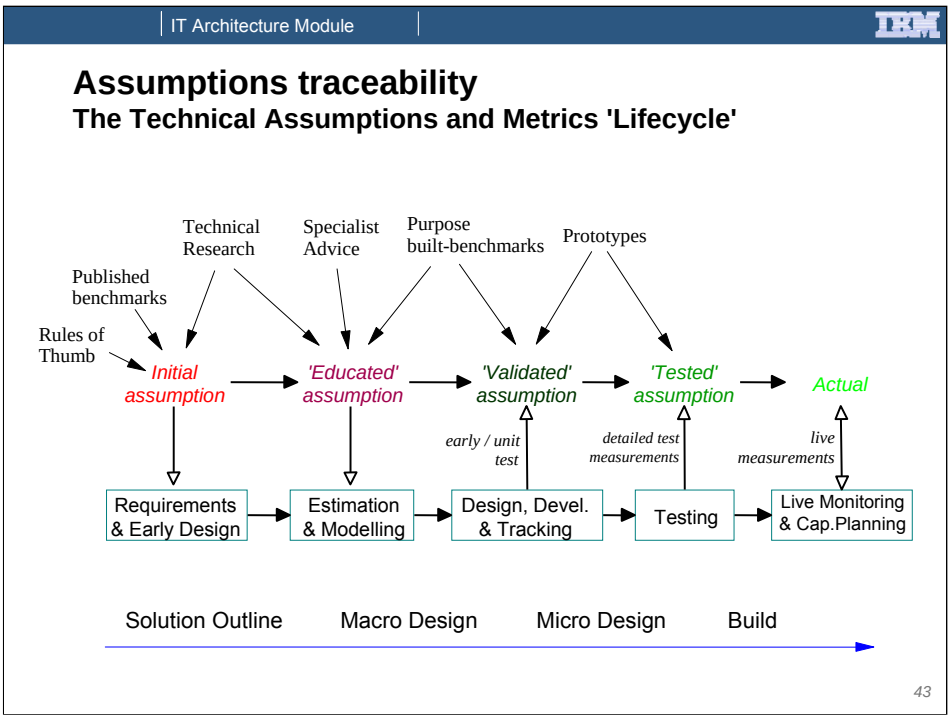
Example M/M/1 queue calculation

Requests arrive at the rate of λ per second	λ = 10 / s
Average service time is Ts	Ts = 0.08s
Server utilisation U	U = 10 * 0.08 = 0.8 (or 80%)
Average queuing time formula for M/M/1 queues: Tw = Ts * U / (1-U)	Tw = 0.08 * (0.8 / (1 - 0.8)) = 0.08 * 4 = 0.32s
Average Response time Tr = Ts + Tw	Tr = 0.08 + 0.32 = 0.40s
Average population in the system (Little's Law), N	N = 10 * 0.40 = 4.0

42

Example M/M/1 queue calculation

The steps in the slide should be self explanatory. In the 4th step, we are using the equation for calculating average wait times for single-server M/M/1 queues where exponentially distributed interarrival times are assumed. Because of the high utilisation in this particular example, the wait time (0.32s) ends up being significant with respect to the service time (0.08s).

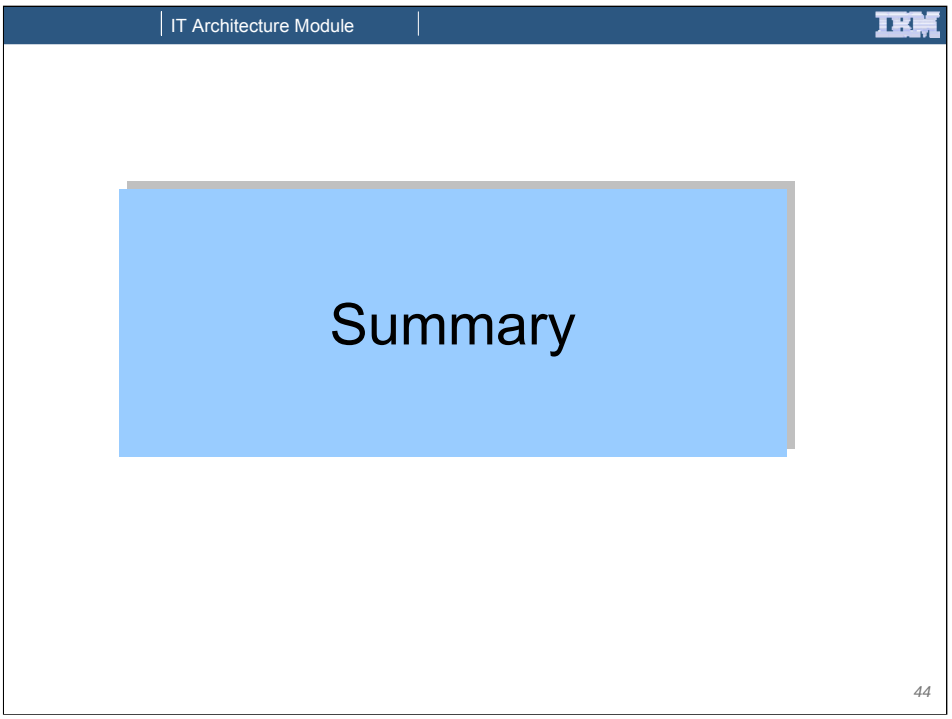


The Assumptions Lifecycle

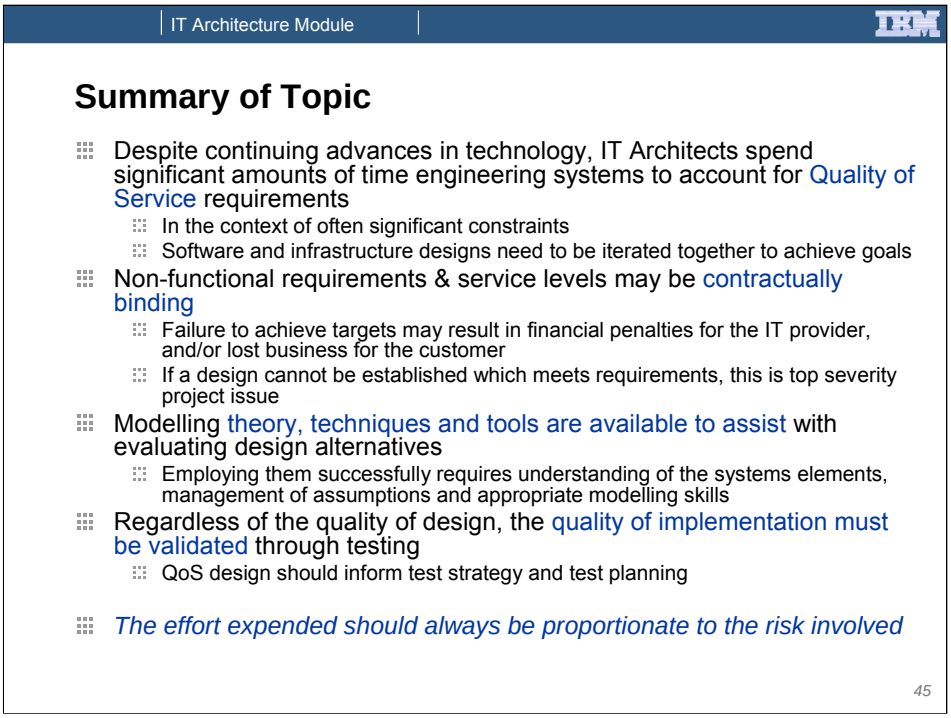
This reflective slide brings together a number of themes from the session and is provided at no extra cost.

In Quality of Service Engineering we rely heavily on assumptions and data which we use to feed our models, inform our engineering strategies and guide our testing and planning for live operations. It is a truism of all complex endeavours that we know more as time proceeds however so does the impact of discovering an unpleasant fact. Therefore the quality, or correctness, of the assumptions we are working with is paramount to the success of our engineering activities.

Taking “assumption’s eye view” of the world, the slide shows how we can use different techniques and activities through the phases of a project to iteratively improve our assumptions and hence our models. It might seem of little value to have a ‘perfect’ model of a system just before or after it goes live, however bear in mind that future business volumes are still unknown; that change the infrastructure is evitable; and that’s there generally a Release 2.0 round the corner ….



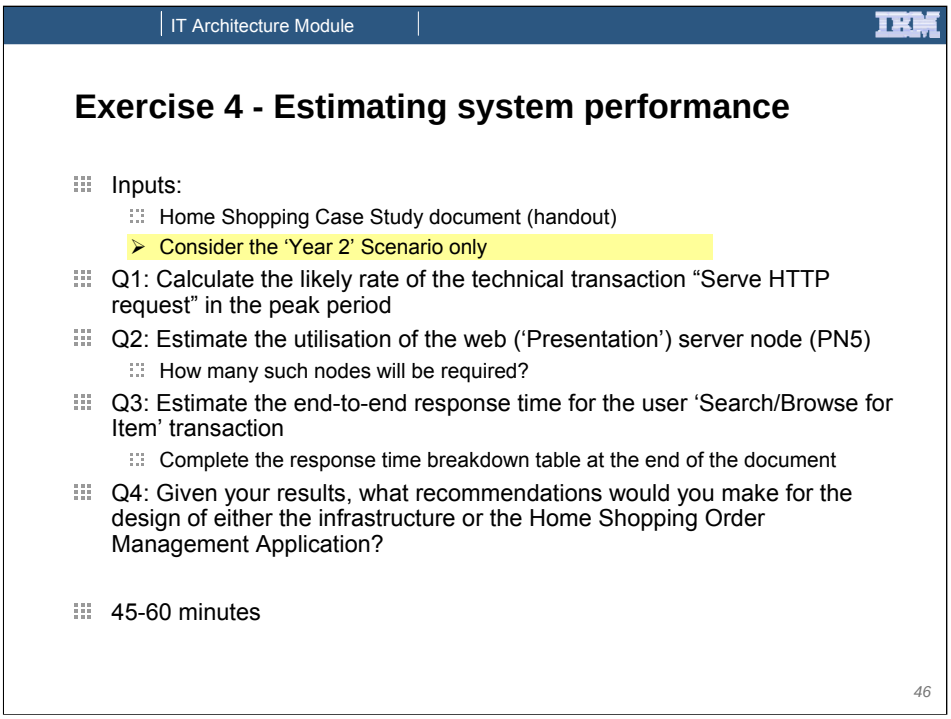
Summary



Seminar Summary Points

The final slide makes the point that technology advances at a tremendous pace (standard components become more reliable; processors and networks become faster; disks increase in capacity, …) however to date this has not put Availability, Security, Performance and Systems Management Architects out of business yet. In fact, technology complexity and the ever increasing demands and expectations of businesses and indeed the general public of IT systems and technology means the quality of service engineering disciplines are just as relevant as when whole bank used to be run on 2 MHz processors accessing 4MB of RAM.

Ultimately, it should be pointed out that all of our efforts should always be aimed at implementing *successful systems* and supporting the delivery of *successful projects*. The tools and techniques we have briefly touched upon in this seminar are means to an end and we must always bear in mind that the effort we expend should be proportionate to the value and risk of the project with which we are involved.



Exercise 4 – Estimating System Performance

Exercise 4 is based upon a cut-down case study document which is separately provided as the Word document "QoS Seminar - Exercise 4 input v*n.m*.doc" and this should be distributed to students if Exercise 4 is going to be attempted. Our experience of teaching this seminar to date is that 3 hours is only just sufficient for giving the presentation materials we have covered so far and conducting the 3 previous in-class exercises. Therefore it is recommended that Exercise 4 be given as an after-class assignment.

A sample answer is also provided in the form of an Excel spreadsheet – the name of this file is "QoS Seminar - Volumetric & Performance exercises full solution.xls". This is for the teacher's reference for both Exercise 3 and Exercise 4.

Basis: Slides 39 – 42.