Table of Contents for Chapter 3

Part 1. Architecture 101 – 'The Language We Speak'

<<< … >>>

Chapter 3. Quality Measures for Your Designs

If Enterprise Architecture is such a big and complex thing, can we measure its quality? We can. And we do measure the quality of the architecture while designing new solutions, or evaluating and comparing existing ones for fitness to the business challenge at hand.

Quality Measures, or criteria of the solution quality, may be qualitative or largely quantative.
If we can attach simple number to the criteria – we are in luck – we've got easily measurable, quantative criteria. Don't despair though. We can always attach some rank or mark to the largely qualitative criteria. Either way, these are measures that you always keep in mind while devising your enterprise solution.

We provide common sense explanations to major *Quality Measures* or Criteria for the Enterprise Architectures.

Constantly, you will face a challenge of analysing the quality of some architecture solution, or evaluate and compare some technologies or products against each other.
You will create a table or evaluation spreadsheet where you capture results of your analysis for each criterion or *Quality Measure*.

You can always apply to the solution *Quality Measures* described in this chapter. Only thing that will change will be importance (weight, or rank, or priority) that you attach to the particular Quality Measure depending on Business Requirements.
After all, if you've got a small customer base and volumes of transactions, you would not worry about *Scalability* that much. Or, if your transactions are highly confidential and critical for business, you will especially emphasise *Security*, *Business Continuity* and *Data Integrity*.

Business Continuity and Disaster Recovery Planning (DRP)

By *Business Continuity* we mean resilience of the Enterprise Architecture to any reasonably conceivable adverse impacts, without any (or at least significant or catastrophic) disruption to the core business.
DRP means roughly the same, but this term is out of fashion.

Business Continuity is always about the fine line between the perceived risk of failure and the mission-criticality of the IT solution to the business (or, in other words - the cost of the IT solution's downtime against the possibility of this happening).
For example, we can survive if the weather satellite images in the application feed are one hour late on a sunny day. But bookmaking business may sustain critical losses if betting server is down before the most important horse race of the season.

Time to Market

When business gets an idea of using some IT solution to achieve its business goals, that IT solution always needs to be available yesterday.
Time to Market is the time span from the inception to deployment of the IT solution, or to the shipment of the solution into sales channels (in case of business being a software vendor).

Pressures of getting the product out may influence the whole dynamics of the IT solution delivery.

We have to balance the quality and the immediately available, 'day one' functionality of the product against the risk of being late to the market and losing the market share, profits and the business itself. Although, compromised quality may negate the short-term wins and increase the costs of support and maintenance of the product.

Total Cost of Ownership (TCO)

This section is really a placeholder for the entire costing and budgeting process, and for other monetary metrics.

We introduce here financial and 'bean-counting' considerations into overall list of Quality Measures for the IT solution. We identify some ingredients that contribute into the total cost of the Enterprise Architecture – what business pays out of its purse for our pleasure of tinkering with exciting technologies.
Well, we are not that insensitive to the pressures of the Balance Sheet or the Profit and Loss Statement – and remember, our professional success and job security directly depends on the success of the business that we serve.

These are very important and pragmatic considerations. Nice technical idea will not take off, if we cannot provide a clear business model and approve a sufficient budget for the project.

Some major ingredients contributing into the TCO are:

- *Infrastructure costs* to support the IT solution throughout the whole life cycle; hardware and software licences costs, for production, staging, testing, development and any other environments as required
- *Labour and Materials* throughout the whole life cycle; salaries and expenses of the employees and consultants, office space and office equipment, consumables – throughout analysis, design, development and integration, deployment, maintenance and support, decommissioning
- *Fixed Assets* for the duration of the project; cost of equipment and its write-offs over the period, real estate and buildings
- *Intellectual Property*; patents and copyrights

Return on Investment (ROI)

How much of the investment is too much? This is relative and depends on expected returns and the value of achieved results to the business, measured against the incurred expenses.

Return on Investment (ROI) puts things into prospective and shows how well the investment into IT solution performed.

Skills Availability

Different skills will be required on various stages of the project life cycle, and for different kinds of the Enterprise Architecture that may employ different technologies and products.

Skills, or the required level of skills may be in short supply, either at the time when project requires them, or in general. In the end of the day, skills shortage translates into project delays and costs blowout – if you want a rare gem, you have to pay for it through the nose, when and if you find it.

Enterprise Architect can manage *Skills Availability* by designing the Enterprise Architecture to open and commonly accepted in the industry standards and APIs, and by containing the complexity and proliferation of technologies and products on the project.
Also, architect should be mindful of existing skills and resources, readily available in the enterprise or on the market (for both development and support); as well as existing infrastructure, products and components in the enterprise. In other words, IT solution will promote re-use of valuable skills across the board in the Enterprise Architecture.

Scalability

Scalable architecture should be able to easily adapt to the increased workload and continue to support business successfully.
We may require scaling IT solution up if customer base increased (which is a good thing), and when users become more active and conduct more business transactions more often bringing more revenue (try to argue against that!). Also, our business transactions themselves may offer over time more functionality and become more resource consuming.

In Enterprise Architecture, we plan for horizontal and vertical *Scalability*.

Vertical Scalability means improving performance and throughput by brute force.
For example, we may have to replace existing server by more powerful model or by upgrading the server with more CPUs, cache, memory or storage. Or, we may replace the network connection between two hubs with the 'fatter pipe'.

Horizontal Scalability is achieved by adding more processes, servers or communication lines – to provide increased overall throughput and better sharing of the workload between various components. *Horizontal Scalability* implies some kind of resource management and *Load Balancing*.

Load Balancing

Load Balancing is the process of sharing workload between two or more components (resources, or service providers) that are deployed to perform some similar tasks concurrently. Resources are pooled together under the control of some management system that performs the role of distributing incoming workload between the available resources from the pool.

Load Balancing is the widely used technique for improving the overall *Availability* and for scaling up the throughput of the service.

We provide examples that explain Load Balancing scenarios later in the book.

Availability

Simply put, *Availability* is when system is there for us when we need it.

Failover – 'come back quick, we forgive everything'

No system is 100% available all the time. Failures happen.

Failover - ability of the system to come back online quickly after the failure - important feature of *Availability*.

Fault Tolerance and Single Point Of Failure

Fault Tolerance is the ability of IT solution to keep service up (at least partially, at least the most important and business critical functionality) under failures, stress and possible abnormal conditions caused by human errors, malicious activities or events of nature.

Failure of some components may not cause a catastrophic failure of the whole system. However, if the failure of certain component stops the show, we say that this component is the *Single Point of Failure*. Enterprise Architect tries to identify all *Single Points of Failure*. We then consider re-architecting the solution to eliminate the *Single Point of Failure*, or to ensure sufficient reliability of this component.

High Availability – '24x7'

High Availability is the collective term for products (both hardware and software) that are dedicated to achieving the extraordinary reliability and uptime of the IT solution.
High Availability is so important, and specialised tools and techniques for achieving *HA* are so different that we distinguish it from the *Availability* in general.

We achieve *High Availability* by clustering and load balancing of separate resources and by mirroring, replication and propagation.

Denial of Service Attack – 'stuff it'

Denial of Service is probably simplest and most common way of inflicting damage on the Enterprise Architecture. This malicious activity is easily perpetrated, and cannot be easily detected and protected against.

Problem is that *Denial of Service* may be caused by perfectly legitimate business transactions – just too many of them.

Most common and well-publicized viruses or worms are in essence a *Denial of Service* attacks on our mail servers.
These programs get hold of existing distribution lists on the mail server by exploiting some security hole in the product, and send e-mails to all destinations, and so on...
These e-mails may contain viruses, but may be a seemingly innocent junk mail. Does not matter – damage have been done already. Mail servers suffocate under workload, hang and go down.
This may happen to any other publicly available or Internet-facing application.

Mean Time To Failure

Dry statistics tells us that system can only be less reliable than the least reliable of its parts. This not necessarily true in complex Enterprise Architectures.
We can build resilient architectures that may keep the service up to the delight of the business even if some components are unreliable and failed.
And even when business is affected severely, we still may be able to maintain some really mission-critical functionality - either in full, or scaled down without the really catastrophic impact on business.

Having said that, even with most resilient systems – if it is broken, it is broken.

There are important statistical measures that provide you with rigorous qualitative measures of system availability.
As an example, Mean Time to Failure gives us an estimate, how long we expect the system to function correctly after latest failure is fixed. Good to know, but as an Enterprise architect you will take these numbers with the grain of salt.

Performance

Performance is a very generic notion and rather widely abused term, and intuitively quite clear.

We consider system to be well-performing if it comfortably handles required volumes of workload and capacity with some 'grunt' to spare, and supports or facilitates our business transactions in a timely fashion, with the efficient use of budget and resources.
This general definition of *Performance* needs to be qualified by more specific and quantifiable criteria and *Quality Measures*.

Latency

Latency (or *Response Time*) defines how much time the client process has to wait for the response back from the server to complete a certain function.

Workflow or the control flow of every business transaction may be viewed as a sequence of smaller activities, or sub-tasks, or links in a chain. Activity may depend on completion of some other activities, and each takes time to complete. You may and will have some measurable delays in execution of activities, and these delays sum up in a whole transaction.

We have to dissect the control flow of the transaction. We can attack the problem of improving the *Response Time* only when we understand what components and layers are involved in the execution of transaction, and what portion they contribute into the overall *Latency* of transaction.

Customer sitting at the browser is not interested in excuses and complex technical considerations. You will not be able to find compassion in salesperson that has seen his deal falling trough because database indexes do not ensure adequate performance of your database queries.
If customer has a choice, this experience can put him or her off for good – try to get your business back then. All of us, Internet users, shall find this scenario very familiar.
This is the bottom line. If the *Latency* is to high to support transactions in the core business, IT solution may be rendered useless.

Capacity

Capacity Planning is one of the primary concerns of the Enterprise Architect.

We have to ensure that business requirements are translated into correct estimates for raw CPU power, memory and cache, disk storage volumes (including mirroring and RAID striping), secondary storage (tapes, cartridges, CD etc), bandwidth of network connections (or thickness of our ‘pipes’), throughput of the message bus.

Bandwidth

Term Bandwidth may be used in different contexts.
Bandwidth or throughput means thickness of our communication ‘pipes’, or how many transactions system can handle in a timeframe.
For example, certain modems can handle transmissions of up to 56K bytes per second – you would not enjoy watching videos over this line very often. Or, server in a banking application may handle 400 transactions for withdrawals from ATMs per second.

<< We give examples of various measures for the *Bandwidth* and *Capacity*. >>

Database Integration

Enterprise Architecture, more likely than not, will require access to large volumes of data, possibly stored in different databases and managed by different database engines.

Enterprise Architect designs the IT solution to ensure efficient access to these databases in the first place (possibly by reconciling different technologies or products idiosyncrasies).
Having done that, Architect looks into ways of maintaining the Data Integrity in all possible database usage scenarios.

Legacy Systems Integration

Complex architectures will rely on some pre-existing components and applications.

'Legacy' implies that this system is somehow obsolete, a legitimate target for scorn, and in the end of its life cycle. Often, nothing can be further from the truth. Core business may rely on such a system after years of fine-tuning and pain, and will rely on it for years to come.

Legacy system in the Enterprise may be entrenched in its day-to-day operations, and business finds it difficult to justify a move to other, more flashy and fashionable system or technology.
This could be costs, time, efforts, skills, resources, insufficient additional benefits to justify the trouble, technological risks and disruptions to the core business, or all of the above. Or, this could be habits, tastes and opinionated views of the senior management.

Either way, *Legacy Systems Integration* is a common challenge of the Enterprise Architect.

For example, building the Web front-end to the existing back-end application provides a typical example of *Legacy System Integration*.

As a *Quality Measure*, *Legacy Systems Integration* defines our ability or existence of the proven patterns for making existing application and the new product or technology work together.

System Management, Maintenance and Support

Support, Maintenance, System Management, Audit and Monitoring account for the sizeable part of overall costs and efforts in the life cycle of the application.

Quality of Documentation – 'if only I knew …'

Absence of the good quality documentation for the IT solution is one way of achieving the job security, in the short term only.

Given the complexity of Enterprise Architecture and the high value of know-how, and that many different teams and various stakeholders are engaged throughout the application life cycle, availability of good documentation is a must.
Documentation ensures continuity of skills and knowledge (especially when knowledgeable staff moves on), and captures the evolution and the current state of IT solution.

Some *Agile Methodologies* (including *eXtreme Programming)* favour cross-pollination and spread of skills and knowledge in the team, often at the expense of voluminous documentation. As usual, truth lies somewhere in between the two extremes.

Concurrency

Modern distributed applications with large customer base of users, who are computer-savvy and proficient Internet-surfers, often presents special demands on our ability to handle multiple concurrent sessions, threads, connections, transactions that compete for the limited resources on the server.

Insufficient capacity to handle concurrency in every link from the user device to the server will result in increased latency and negative user experience at beast, and system lockdown at worst (possibly affecting other critical applications as well).

Good concurrency metrics – something we have to design to from start and configure or build into our applications. Unit testing of the application will not identify potential concurrency problems – you have to rely on specific Stress and Volume Testing (SVT) to address these concerns.

Concurrency improves overall utilisation of resources, throughput and user experience. At the same time, concurrency brings its burden of higher complexity 'under the hood' and possibility of two concurrent processes clashing when trying to get hold of some shared resource (database, memory, CPU etc).

Capability for *Concurrency* (apart from, at to the lesser extent due to the brute force of our hardware) is determined by ability of the system for multi-threading and multiple connections.

Threads

Program executes a single thread of control (or control flow) that corresponds to a single-threaded process.
In order to achieve concurrency, several threads may control execution of several processes simultaneously.

Each thread will have its own context of execution, at least logically. However, we have to be mindful which limited physical resources still remain shared.
For example, we can run several applications and keep several windows open on our PC. However, PC may have just one CPU. Physically, processes execute sequentially, having an audience with the CPU in turn. It happens so fast, that we just have an *impression* of the concurrent execution. Not the first time that computer manipulates our perceptions - we cannot help ourselves and let the computer cheat.

Connections

In order to communicate, processes have to establish *Connections*, and to start the conversation using some agreed protocol.

By *Connection* here we mean network connection, database connection, or any virtual connection from one process to another in general.
On the lower level of abstraction, such virtual connection translates into logical API or ORB message bus, database connection or physical network connection.

Connection is an expensive limited resource. *Connection* presents a window into some other world with its own attractions – server IP address and port that some application listens to or database connection with the goldmine of valuable data.

Common pattern in the Enterprise Architecture is to ensure that *Connections* are established and opened beforehand, and amassed into the *Connection Pool* for the taking by the processes as required. 'As required' is the operative keyword here. Being an expensive resource, *Connections* must be released by the process back to the pool immediately. Well-behaving processes will avoid hogging the *Connection* – selfish process itself would be all right, but his humble processes-peers are going to wait in line and suffer.

Separation of Concerns

Programmers knew long ago that breaking the demarcation lines between logically separate fragments of the program by creating the lattice of cross-references is no good.
This creates a 'spaghetti code' that is difficult to build, test and maintain. Computer would not care – it would run without qualms any dumb program, as long as it is tested and let alone. However, program has its life cycle and needs to be worked on by people from time to time.
Strange as it sounds, program is written for people, and people are the weakest link here.

Psychology indicates that people cannot keep in their mind and actively consider more than seven things at a time. This is the physiological limitation of our thinking processes on this stage of the evolution of our species.

This limitation holds true for any type of human activity, and the Enterprise Architecture is no exception.

We manage complexity in all our endeavours by conscious and consistent application the *Separation of Concerns* principle. We break up complex problem into smaller separate pieces and attack them in our further analysis one by one. If these smaller pieces are still too big, we know what to do…

Separation of Concerns helps us to identify components and layers in our architecture. However, benefits of the *Separation of Concern* do not stop and the making complex system more comprehensible. It allows achieving smooth integration of various products and technologies by orchestrating their interoperation through clear and stable *Integration Interfaces*.

Modularity and Layering

Separation of Concerns in the Enterprise Architecture manifests itself through *Modularity* and *Layering*.

Modular design identifies individual components or entities – nuts and bolts of the architecture – and ways they communicate with outside world and each other.
In terms of Object-Oriented Analysis and Design (see later in a book), as far as Enterprise Architecture concerned, components are being defined through their internal state and behaviour in communicating with the outside world.
Behaviour of the component or entity is exposed to the outside world through the *Published* or *Public Interface*, and only through the interface. Components do not (or rather should not) have a hidden agenda or secret ways of communicating with the world, not defined in the *Public Interface*.

Layering means some logical grouping of similar components – either by their common high-level function in the architecture, or stage in the control flow, or level of abstraction, or level of granularity and detail of our components.

Cohesion

Cohesion in architecture means that components and processes, products and technologies are not too different, so that architecture looks like a patchy quilt.
Wherever you find a rough seam in your architecture, expect it to cost you at some stage, literally - round peg was not meant to fit into the square hole.

Loose Coupling and Tight Coupling

Loose Coupling of components diminishes dependency of one component from another. And this must be a good thing, because that is why we tried to define our components in first place.

Tight Coupling means that some components, products and technologies depend on each other – you change one component, and it may break something in another, and the whole house of cards may collapse.
For example, we suspect, that the Internet Explorer browser is *Tightly Coupled* to Windows Operating System. Some people like it.

Insulation and Encapsulation

Hiding the implementation of the component behind the interface is the reliable criteria of good system design.
We call it *Insulation* and *Encapsulation*, meaning that ideally outside world should not care how component is implemented as long as it complies with the public interface (or its contract with the world).
If we achieved that component can be replaced by another one without the world noticing.
For example, we can replace component by the product from another vendor, or rewrite component so it performs better.

Usability and User-Friendliness

We build computer systems to serve people, not other way around. And we do not build systems to satisfy bruised egos of some computer nerds… Well, not every time.

It is widely under-appreciated notion that computer system should go out of its way to bring a satisfactory user experience. If users use the system just because they must, they will get you back eventually by finding way around or by not using your services next time.
You should clearly understand who your users are and what level of their proficiency you can legitimately count on.
Different system failings may cause the negative user experience – slow response, complexity of use, poor graphics and presentation, misalignment with normal business processes (or with user's habits and tastes) etc.

Accessibility

Accessibility ensures that computer system can assist to and cater for needs of people with various kinds of disabilities. Some computer users may be visually impaired, some may have difficulty to use a keyboard, some may have other medical conditions that prevent them making use of computer, or some of its equipment, or particular program.
Don't forget all people are equal, and anyone can become a paying customer – do not disappoint or upset them.

There is multitude of devices and programs that make life of people with disabilities easier.
Also, there are industry standards and legal regulations that make certain accessibility features mandatory.

Re-use

Re-use implies that some component is used more than once, and some savings and other benefits have been achieved in the process.
We can re-use any artefact of the Enterprise Architecture – hardware, programs, skills, patterns etc. We should strive to do just that.

Immediate pressures of the project will act against our best intentions for re-use. After all, who cares what happens next if project is not delivered on time and on budget, and heads will roll. True, but not quite.
Sometimes some 'professionals' may resist re-use due to own agendas like job security. Any aspiring Enterprise Architect must understand, that nothing can be discredited easier, and nothing will let you reap more rewards in the long run than your professionalism and professional integrity.

Re-use may require marginally greater effort first time around, but most likely the potential benefits of re-use worth the trouble (and not much of it anyway – our design guidelines and delivery methodologies are primed for re-use very well).

Maturity

It is very exciting time in IT industry. New technologies and products appear constantly and evolve rapidly. No wonder, there are many fads and broken promises, even with the best intentions.

Enterprise Architect shall keep very cool head and be able to position products and technologies correctly, and to take measured and well-informed risks when necessary.
Maturity of the technology or product manifests itself in many dimensions – broad customer base and positive references, stability and support of the vendor company, time span on the market, clearly articulated roadmap, compliance with forward-looking technologies and industry trends etc.

In a rapidly changing technological landscape, picking the mature winner with the staying power is still more of the art and a gamble. Often, there is just not enough history in the promising new technology or product – Enterprise Architect has to make a judgement call based on experience and professional insight.
Diligent analysis, expertise and mature attitude of the Enterprise Architect himself or herself have no substitute in finding the good workable and feasible solution in the complex architecture. Note, we do not aim for 'perfect' solution – there is no 'perfect' solution, as there is no black and white in a complex architecture for the large enterprise where we are trying to find an elusive balance and hard compromises between sometimes conflicting goals.

One practical litmus test or criterion for maturity may be very useful though, despite its simplicity. This criterion is called 'version 1.0 syndrome'.
Beware version 1.0 products. There may be nothing wrong with this particular product, but chances are that some teething problems will happen. Product is yet to prove itself in the battle.

Viability and Feasibility

Enterprise Architecture solution is likely to have an immense impact on the core business of the Enterprise.

Usually, we deal in the Enterprise space with complex technologies and products, and multitude of them. As a rule, this requires stretching of our human resources, infrastructure and budget. With such commitment and investment, we better get a decent return and value from this IT solution back to the core business.

Viability and Feasibility Assessment analysis has to be done before we invest and commit resources. This analysis has to substantiate the forthcoming decision on investment.

Combination of technical, business and financial considerations together will contribute to the final go/no-go decision on the project.

Security and Privacy

We mentioned *Security* and *Privacy* as a concept before. They are very important Quality Measures as well.
Enterprise Architecture has to provide necessary protection of Enterprise information assets and applications through adequate levels of *Security* and *Privacy*.

Transactional Capability

Notion of the Transaction and its ACID properties have been explained earlier in the Key Concepts. Transactional Capability, as a Quality Measure for the Enterprise Architecture, needs to be mentioned in this section as well.

Data Integrity

Data Integrity is the generic data Quality Measure that covers data completeness, timeliness, correctness and consistency, for the purpose of the application.

Data Integrity may be compromised by physical corruption of bits and bytes on the storage media, or on transmission and presentation of data.
Also, Data Integrity of seemingly benign elementary pieces of data may be compromised due to logical inconsistency as far as some business rules of the application concerned. For instance, if you maintain census profile database, and some 12 years old person was reported as being married, this must ring a bell in respect of Data Integrity, at the very least.

Data loss or corruption may happen for many different reasons - due to technology fault, deficient business logic in the application software, unintentional human error, destructive impact of environment, or malicious human activity.

Data Integrity is of the primary concern for the transaction. Violation of ACID properties for the correctly functioning transaction leads to the corruption of data, or their unavailability in the required timely fashion.
Data Integrity and validity may be governed by very complex business rules. Not all of them can be easily enforced, especially in the 'rainy day' scenario.

Data Integrity violation and corruption of data may manifest itself as a nuisance (misspelled title), or as a catastrophic failure of some mission-critical business function (wrong share price in the large stockbroking deal).

Enterprise Architect has to find an acceptable and practical compromise between guaranteed levels of Data Integrity and the reasonable risks.

Data Integrity features of DBMS are based on maintaining a log of database updates in their chronological order.
Every update is being captured, and the log entry is created with the timestamp attached to it.

If we discovered data corruption, we select the point in past time when we surely had our data in the correct state. We instruct database software to *rollback*, or *restore* database to the state as it was at that point in time. We may have to live with the loss of database updates that happened since that point in time to present.

Standard(s) Compliance

Standard Compliance is the cornerstone for Interoperability. Different technologies and products from different vendors can work together only when builders religiously adhere to the relevant standards.

We distinguish Standards Compliance of technologies (and compatibility of various standards themselves) and compliance of the implementation of the standard in particular product.
Product may be standard-compliant 'on paper', or somehow 'partially-compliant', and it will cause you a lot of grief when you commit yourself and try to make components work together.

We suggest you relentlessly seek assurances of standard–compliance, references for proven history of successful interoperability, and certifications, where available.

Interoperability and Compatibility

Enterprise Architect faces challenges of putting together a large and complex system where every component may have its own idiosyncrasies. Life is not meant to be easy, and some developers make sure it is not.

We are looking at available APIs or adaptors that make components talk to each other. We say that we aim for *interoperability*, and interoperability is achieved easier if components or interfaces are *compatible*.

Portability

Portability in IT means ability of the product or technology to run in different environment or on different platform.
For example, Java technology religiously sticks to the WORA principle, i.e. 'write once run anywhere'. This is achieved by running Java products in strictly standardised Java Virtual Machine (JVM) container, and by ensuring that JVM is available on every conceivable platform.

Enterprise Architectures (due to the sheer size and complexity of business requirements from the various lines of business) are more prone to the mixing of various technologies and platforms, and *Portability* may become and important criteria for some enterprises.

Open and Proprietary

Proprietary technology implies some limitations or additional efforts in a sense of interoperability with other technologies and the level of acceptance in the industry.
One of the dangers of the *Proprietary* technology lies in a future proofing of our architecture (when we need to migrate or integrate our system later) or in the lock-in to the single vendor.

Open standards provide strong arguments for the investments protection.

Future Proof and Shelf Life

Nothing will last forever, and does not have to be. You won't produce a steel table tennis ball just because it cracks when you step on it. Or, to put it other way, we build courses for horses.

Standards change, technology evolves, and business requirements change as well. We build the Enterprise Architecture to address specific needs of the business application, and to be resilient just enough to make sure that we keep service up without major dramas throughout the lifespan of this application.

Extensibility and Resilience of Architecture, 'Architecture Rot'

World is not perfect and constantly changing. In essence, we model real world processes in our IT Systems, and they are not perfect too.

Our IT Systems are not perfect for many reasons:

- Model can never be as complete as the infinitely complex real thing. At least in the area of Enterprise Architectures, we will not be able to automate completely the business processes for some time yet. There are domain experts or users, who will take our results with the grain of salt. We capture main target features for the Enterprise Architectures through Business Requirements and, inevitably, something will be incomplete or lost in the translation.
- We always have limited resources, ability and time in building a perfect IT System. If we spend too much money and time to deliver a lot of high-quality functionality, this would not be a perfect (to say the least) solution for Business – too late and sends the Business to the wall. Often, Enterprise Architecture has to make a series of pragmatic compromises.
- Real thing changes on us and we always have to play a catch-up game.

These are forces that may require many small and large changes and enhancements to the Enterprise Solution throughout its life cycle.

If we did not build our Enterprise Architecture to be *extensible* and *resilient* to the expected changes from start, we often apply patches and band-aids. In a complex system, we soon reach the breaking point of our ability to comprehend and manage the system processes 'under the hood'.
New changes introduce unexpected problems elsewhere. Fixes introduce problems that are worse than original ones. System becomes too expensive to support and basically falls apart. We've got a Big Ball Of Mud on our hands [Foote 2000].

We call this unpleasant situation 'Architecture Rot' – and it really stinks, and costs as well. Also, there is one insider's joke referring to 3-*tear*, as opposed to 3-*tier*.
Bad news – this may happen to you, good news – you've been warned. We are just kidding, of course.

Point we are trying to make – strive for the 'good' over-arching reference architecture with clear separation of concerns and clean public interfaces or contracts between components and layers, to

satisfy Quality Measures for your Enterprise Solution described in this Chapter. Stick to this reference architecture throughout the life cycle of the Solution. You will be glad you did.

<<< … >>>