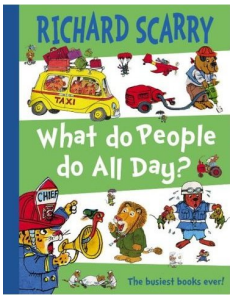
University Relations

What do IT architects do all day?

December 2006

Contents

1 Introduction



Aficionados of American children's literature may recognise in the title of this lecture a reference Richard Scarry's *What to people do all day?* [1], a childhood classic from the 1960s that remains as delightful as it is dated. It is a book that explains how things work, and the part people play in keeping them working. It is full of cut-away drawings and unexpected connections that explain how water, power, raw materials, food, and transport systems are provided to keep an overall system working. That system is of course the community in which we live.

Without wishing to strain the analogy too far, it is possible to see in Scarry's work the childish beginnings of one of the core motivations of the IT architect: a desire to understand not only how the individual details work (at lease in concept) but also an interest in seeing the connections between things, and the big picture of a busy society at work. For this is — in essence — what the IT architect is about: understanding enough about the details to make informed decisions and seeing the overall situation clearly enough to direct resources to what is important.

> IT architecture [has] proved to be successful in providing an analytical and decision-making framework for a sequence of new initiatives or changes, at the same time ensuring design integrity and stability. — Cherbakov, *et al.* [2]

This document accompanies a seminar that introduces the concepts, roles, and activities of IT architects. This text is deliberately written in a text book or discursive style instead of simple lecture notes: in this way you get a readable reference to which you can return after the seminar in order to explore the ideas in more detail. Suggestions for further study are included in the concluding section.

The material for this introduction to IT architecture session has been adapted from IBM education material that is designed for fairly recent graduates who have had 18–24 months experience on IT development and implementation projects. The IBM material comes from a class that introduces a broad sweep of technology from the basic hardware components of computers, disks, and networks, through operating systems, databases, and middleware, up to application development and deployment environments. The emphasis is on the principles of how these things work, rather than on the specifics of the current implementations available in the market; these are what is important for a broad understanding of IT (also they change more slowly, so it is easier to keep the class material up to date). The class is taught in three or four days and intentionally covers a very wide range of topics, in order to help students appreciate the 'big picture' of technology and to make connections from one topic to another.

This ability to connect and to have insight into the general principles of communications and information technology is one of the key characteristics of a good IT architect. This introduction is designed to show you what other characteristics are required in an architect, to give you an understanding of the issues that they wrestle with, the roles they play on projects and in IT departments, the vocabulary, techniques, and methods that they use, and the different career paths that they might follow. Your learning objectives for this session are to appreciate what IT architects do and why they are needed.

⋆ ⋆ ⋆

In order to cover the ground at a fairly brisk pace, we make some assumptions about you: first that you are studying a relatively technical degree subject and that you already are familiar with the basic ideas and concepts of communications and information systems; second that you have an interest in the application of technology to business problems (for that *is* what architects do); and thirdly that you have at least a passing familiarity with the organizational structures (projects, operational departments, development teams, etc) that are found in the IT industry, preferably from some first hand work experience. At least two of these three are required for you to get the most from this course, but please do not be afraid to ask and to question what is said if there is anything that is not clear, or that contradicts things that you thought you knew.

In preparation for this course you were asked to review — at least to skim-read — some Gartner Group articles [3, 4] about IT architecture. We will not formally review these papers, but we will refer to them, and you are welcome to raise issues about the matters that they discuss. You are warmly recommended to re-read them after the course.

This session will consist mainly of lectures, although there will be some exercises and group discussion. Your active participation is requested during these exercises and discussions; afterwards your feedback; and at all times, your attention.

2 Concepts and ideas

We are all familiar with the term architect in its traditional sense, a designer of buildings, bridges, or gardens. More an artist than an engineer perhaps, but how does the term translate into communications and information technology? Thirty years ago, technical staff at IBM were called ‘system engineers’, and their job was essentially to help customers install and use computer equipment; as computing has evolved this job has also evolved into what we now call an IT architect.

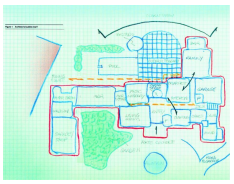
2.1 The idea of architecture in IT

There are two distinct usages of “architecture” in computing: of these one is rather more interesting than the other. If we look at the dictionary definition below, we find the less interesting meaning at sense **4**:

> **architecture** *noun* **1** the art, science and profession of designing buildings, ships and other large structures and supervising their construction. **2a** a specified historical, regional, etc style of building design, especially when it is thought of as fine art • *Victorian architecture*; **b** the buildings built in any particular style. **3** the way in which anything is physically constructed or designed. **4** *computing* the general specification and configuration of the internal design of computer or local area network. **architectural** *adj.* **architecturally** *adverb.* ETYMOLOGY: 16c: from Latin *architectura*.
>
> — Chambers Online

We will come back (briefly) to sense **4** when we discuss the parts of the IT architect’s job that deal with infrastructure, but it is not the main purpose of this course. (If you thought

you were going to find out about the architecture of chip design, you have been sadly misinformed; you had better go to the library and get a copy of Hennessey and Patterson [5] to read). The more interesting sense is actually the first, when 'other large structures' is taken to include major business applications of computing.



One of the earliest (and still useful) attempts to define this usage in computing was made by John Zachman writing for the *IBM System Journal* in 1987. With the increasing size and complexity of IT systems, Zachman foresaw the need to provide a means to control the division of the system into manageable components, and the integration of these and perhaps external components into a single entity that supported a business need. Borrowing from the construction industry, his paper introduces the idea of different **views** of a system; each one designed to assist those involved in the project to understand what it being built and how their part fits into the whole.

John Zachman has made an entire business out of elaborating his idea (see `www.zifa.com`), but original simple framework segmented a system vertically into data, processes, and distribution; and horizontally into layers of increasing detail, and looked like this:

Framework	Data	Process	Network
Scope description	*Business entities*	*Business processes*	*Business locations*
Model of the business	*E-R diagram*	*Functional flows*	*Logical network*
Model of the information	*Data model*	*Data flows*	*Distributed system design*
Technology model	*Data design*	*Structure chart*	*System architecture*
Detailed description	*Data base description*	*Program*	*Netork architecture*
Actual implementation	*Data*	*Function*	*Communications*

The contents of the boxes were not strictly prescribed, in order to make the framework more flexible (the above contents are examples). The key idea is that at each level the view across the system can be more specialised or detailed, but will remain consistent with the layer above. Zachman illustrated this idea with examples from the construction industry. So at the top level would be the client's view of what they want, while down at the technology model level there might be the electricians' wiring diagrams, and the plumbers' installation plan.

Each level can be associated with activities that you might already have experienced on software development projects. These activities can be grouped into phases.

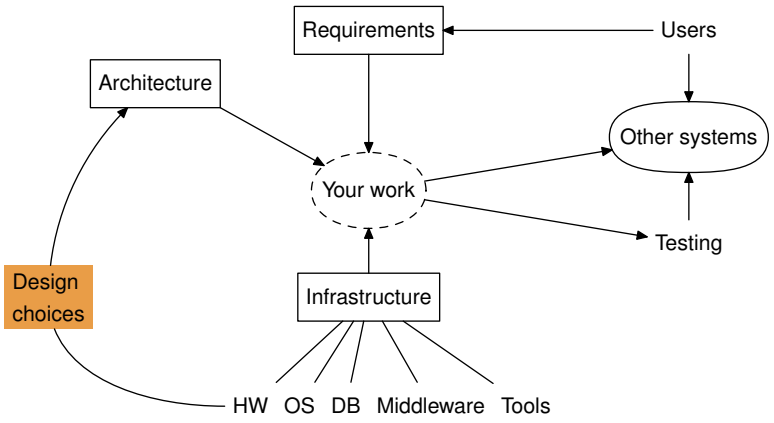
- Working with clients to capture their business requirements
- Analysis of how the users work and how their requirements translate into IT terms

- Design of a whole system, or may be just parts of a system; and the development of high level designs, low level designs and module specifications
- Building software to a specification, developing and running unit tests to ensure that your module does what it should
- Testing a collection of modules that form a complete system

Your experience may also include a spell in a support team that fixed problems after a system went live. Or even part of the team that actually deployed what your colleagues had written. In some projects it is possible that you have been involved in running the development systems, and also running the production systems for real. If you did, this might have included monitoring the performance of the system, fixing bugs, controlling changes, scheduling releases etc. Even if you have never done any of these broader roles, you are probably aware that clients have (large) groups of IT staff who do these things.

2.2 The wider context

If you ever have written software as part of a bigger team, then when you finished testing your modules, you may have handed them on to another team to do integration testing, or system testing as it's sometimes called. Your module would have had to work with other components in the client's system, and the whole system that you and your colleagues had built would have to work with other systems that the client might be already running. To the other systems in the integration test, your part of it is nothing but a black box that takes certain input and creates output — preferably in a correct form. Assembling and testing a large system is easier when all the components have clean, well-defined *interfaces*.



Going back a little, when you designed your part of the system you may have followed design guidelines that constrained how you were able to solve your design challenges, and that shaped the overall result of your design. When you built the system, you — probably without being aware of it — used development tools and a development environment that had been carefully selected long before you started work. These either came with the application package you were working on, or were the client's choice — so that what you created would work with other existing systems.

The important point to note is that there are choices to be made and that the notion of well-defined interfaces is what makes it possible to assemble complex systems. To understand this better we will look at the issue from the other way around.

2.3 What happens next?

In the seminar we do an exercise called “What happens next?” that helps us to understand the layers of complexity involved in any computer system (or any complex system at all) and why interfaces are important.

We take a hypothetical application that does on-line stock checking over the Web. The interface is simple: you select a stock item, you press the ‘check’ button and information about the current level of stock for that item is shown. The exercise explores what happens between the moment you click on the mouse and the moment the information appears on the screen.

☞ *If you have not yet done the exercise, take a few moments to think about what components of hardware and software are involved in translating the movement of your index finger on the mouse into some information in a database.*

The first point of the exercise is to demonstrate that without layers of **abstraction** we would never get anything done. If we had to start every system development by building a pointing device and writing software to interpret the electronic signals that it emits, then we would never get anywhere. This is called reinventing the wheel and you should avoid it; nevertheless it’s surprising how often programmers do it. In an academic environment (such as IBM Research division) re-invention might be a valuable learning experience, or might (just possibly) turn up a wonderful novel technique, but in the commercial world, it is generally preferable to reuse well-known and existing functions or components wherever possible.

Even when there are no components available — because we are genuinely breaking new ground perhaps — it would be important to create some to get our work done. It is a fundamental principle in good design and IT architecture to keep each part of our design consistent and coherent. You are storing up trouble if you mix up the management of your user interfaces with your business logic. This principle is called the **separation of concerns**. If you adhere to it you will be able to test each part independently before you try to integrate the complete system; and when you come to maintain your creation (or when someone else has to) the job will be simpler if you have clearly separated the logical groups of function.

In fact this separation of concerns is also useful in describing a system as a whole or an enterprise-level collection of systems, but on a different level (or perhaps in an orthogonal plane). Clearly the detailed design will be vertically segmented into the different components that make up the whole system, but in documenting the architecture of a system or an enterprise it useful to separate the concerns of those involved in understanding the system too. What we need are different **views** for different people. We will explore this shortly.

The final point that should emerge from the exercise is that just as the mouse cable fits into a standard socket on the back of your computer so the mouse driver fits into a standard interface provided by the operating system, and its signals are presented to application programs in a standard way. All these interfaces are defined **standards**. How they are defined we’ll come to in a minute.

It should be clear that when developing a system there is a great deal going on beneath the surface that you don't see — and that you don't want to see. In computer buzz word terms, it is *transparent* to your application. As an aside, it is interesting to note that there are also things that you do see in your program but that are not really there — memory is the obvious example. The following table may help you remember the buzz words:

	Can see it	*Can't see it*
Can touch it	real	transparent
Can't touch it	virtual	requirement

Now we will look in more detail at standards and views.

2.4 Standards

We are now beginning to look back up at the wider picture, and we will eventually discover that there is the same need for interfaces and standards at that level too. First we will consider formal — and informal — computer standards.

Successful standards are developed in more or less the same way, and not just in the computer industry. They start with a good idea. Someone comes up with a clever way of solving a problem. The solution is normally arbitrary, like making clocks that go clockwise rather than the other way, or laying railway tracks 4 feet 6 inches apart. There are often competing solutions to the same problem — like laying railway tracks at 7 feet and a quarter of an inch apart. These other solutions may be technically superior, and there may be a great debate about them. Manufacturers may make their products to fit one or the other, typically they have to choose in order to keep costs down. Eventually one side wins and the other standard is dropped — even the Great Western Railway eventually took up its broad gauge tracks. Finally the standard may become enshrined in some form of legislation. These three stages in the life of a standard may be called: proprietary, *de facto*, and *de jure*

One could argue that proprietary standards are not really standards at all, but it's important to note that some standards are owned by private commercial organizations and others are in the public domain, controlled by committees and industry representatives. As the long history of anti-trust lawsuits and other legislative action against large IT corporations shows, governments tend to take a dim view when a *de facto*, proprietary standard is used to gain unfair commercial advantage.

It takes a long time for formal independent bodies, like the International Standards Organization (ISO), or the British Standards Institute (BSI) or the American National Standards Institute (ANSI) to act over standards, so in recent years the trend has been for commercial companies to co-operate and form a consortium to build up support for their particular pet standard. In the mean time the market place may have different ideas.

If you browse the catalogue at the ISO web site, you will get an idea of the breadth of what can be standardized. The list includes things that you may not have considered suitable for standardization, for example ISO 12207:1995 is the international standard for software life cycle processes. However the widely used ones nearly always define some form of interface between systems. These include:

- The syntax of programming languages
- Protocols for communication between systems
- Definition of the behaviour of standard Unix tools
- Guidelines for presenting information to users on the screen
- Standards for defining other standards

This last may sound bizarre, but it covers an important area. The obvious omission from the above list is what we refer to as APIs — application programming interfaces — and the reason it is omitted is because they are nearly all proprietary, and although some are in so widespread use that they are standards *de facto*, very few of them are so *de jure*. However there *is* a standard for describing an API. It's called 'interface definition language (IDL)', and it's widely used by certain forms of middleware. Did someone say "but what is an API?"? Here's what the Free Online Dictionary of Computing has to say: See `www.foldoc.org`.

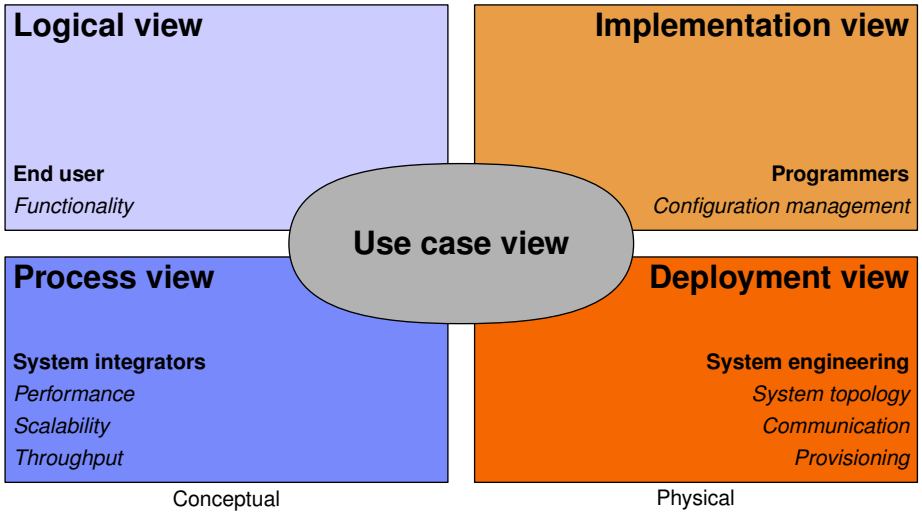
> **Application Program Interface** ⟨*programming*⟩ (API, or "application programming interface") The interface (calling conventions) by which an application program accesses operating system and other services. An API is defined at source code level and provides a level of abstraction between the application and the kernel (or other privileged utilities) to ensure the portability of the code.
>
> An API can also provide an interface between a high level language and lower level utilities and services which were written without consideration for the calling conventions supported by compiled languages. In this case, the API's main task may be the translation of parameter lists from one format to another and the interpretation of call-by-value and call-by-reference arguments in one or both directions.

2.5 Views

The discipline of architecture was invented to cope with the complexity of modern sophisticated communications and information systems. The key aims of the discipline are to be able to present different, but coherent, views of a system in order to allow different specialized teams to work effectively together on a large project.

By treating other parts of the system as black boxes with well known interfaces (but whose implementation details are hidden from us) we can make progress towards building useful functional systems. Instead of dealing with the low level electrical impulses coming from a mouse (for example) we deal with an idealized abstraction of a mouse represented as a device driver. This approach not only makes us more productive, but insulates us from lower level changes. For example we can use a new infrared mouse, or a Bluetooth mouse, without needed to change our application programmes. This simple idea of a dealing with hardware complexity can be extended to help us to deal with similar complexity and change in software.

The following diagram shows a high-level summary of the "4 + 1 Model View" that is a central part of the IBM Rational Unified Process for software design and development.



This diagram represent a system architecture. The use case view in the middle controls the whole picture and the other views are specializations derived from it. A 'use case' is a *written down* document that describes one particular interaction with the system. Use cases involve actors (people or machines or other systems), processes, inputs, output, sequences, and timings related to some business task. They are grouped in related sets and may have alternatives paths either surrounding them or embedded in them. Together the use cases define the system. They are useful in many ways. They can be easily understood (with a little training) by ordinary users and business analysts. They are (supposed to be) a succinct and unambiguous way of writing down the business requirement for what the system actually does. They can be used to test the system on paper and in early prototypes, and so on. Alistair Cockburn's book *Writing Effective Use Cases* [6] provides a good practical introduction.

The other four views are aimed at the various audiences that need to understand the system, but they are not intended to stand alone in all cases. Programmers are allowed (even required) to look at the other views in order to complete their work. Nevertheless this separation of views allows for a more complete approach to the documentation of the architecture and makes it easy for large teams to coordinate their work.

There is no room in this class to do more than introduce these views here, but this area is well provided with good references. In particular you should read *Documenting Software Architectures* [7]. The author of the 4 + 1 model, Philippe Kruchten from IBM Rational heartily recommends it. The following is from his online review [8] of the book, and explains some of the background to why they invented the model.

> Ten years ago, I was brought in to lead the architecture team of a new and rather ambitious command-and-control system. After a rocky beginning, the architectural design work started to proceed full speed, with the architects finally forging ahead, inventing, designing, trying, and resolving, in an almost euphoric state. We had many brainstorming sessions, filling whiteboards with design fragments and notebooks with scribblings; various prototypes validated–or invalidated–our reasoning.

As the development team grew in size, the architects had to explain the principles of the nascent architecture to a wider and wider audience, consisting not only of new developers but also of many parties external to the development group. Some were intrigued by this new (to them) concept of a software architecture. Some wanted to know how this architecture would impact planning, organization of the teams and the contractors, delivery of the system, and acquisition of some system parts. Some parties wanted to influence the design of this architecture. At a further remove from development, customers and prospects wanted a peek, too. So the architects had to spend hours and days describing the architecture in various forms and levels and tones to varied audiences, so that each audience could better understand it.

Becoming such a center of communication slowly stretched our capacity. On one hand, we were busy designing, and validating the architecture; on the other hand, at the same time we were communicating to a large audience what the architecture was, why it was the way it was, and why we did not choose some other solution. A few months into the project, overwhelmed, we began having a hard time even agreeing among ourselves about what it was we had actually decided.

This led me to the conclusion that 'If it is not written down, it does not exist.' This became sort of a leitmotiv within the architecture team for the following two years. The architecture could have been whatever we had talked about, argued, imagined, or even drafted on a board. But in the end, the architecture of this system was only what was described in one major artifact: the Software Architecture Document (SAD). Architectural elements and architectural decisions not captured in this document simply did not exist. This one rule — 'If is not in the SAD, it does not exist' — provided incentive to evolve the document and keep it up to date, almost to the week. It also gave us an incentive not to include anything and everything, such as untried ideas, in the SAD, which became the project's definitive arbiter and a central element in the life of the project. It was our display window for showing off our stuff, our comfort when we were down, and our shield when attacked.

The key questions we faced at the time were: What should we document for our software architecture? How should we document it? What outline should we use? What notation? How much or how little information should we include? There were few exemplars of architectural descriptions for systems as ambitious as ours. Driven by necessity, we improvised. We made mistakes, and corrected them. We rapidly discovered that architecture was not flat, but rather multidimensional, with several intertwined facets. Some facets — or views — were of interest to only a few parties. We found that many readers would not even open a document that weighed more than a pound, and that we would have a hard time updating it anyhow. We realized that, unless we captured the reasons for our choices, we were doomed to reconstruct them again and again, every time a new stake holder with a sharp mind came around. We picked a visual notation that was neither too vague and fuzzy nor too esoteric and convoluted, to avoid discouraging most parties.

Today, software architects have a great starting point for deciding how to

document their software architecture: With this book, you will have what you need in your hands. The authors went through many experiences similar to mine and extracted the important lessons learned. They read many software architecture documents. They reviewed the academic literature, studied all the published books, checked the standards, and synthesized all of this wisdom into this handbook, which describes the essential things you need to know in order to define your own software architecture document. You will find guidance for defining the document's scope and organization, and on the techniques, tools, and notation to use (or not to use), as well as comparisons, advice, and rules of thumb. You'll also find templates to get you started, and continuing guidance for the times you get lost or feel despair along the way.

This book is of immense value. Description and communication about a software architecture is crucial to that architecture's many stake holders. This handbook can help you with both and save you from months of trial and error, lots of undeserved hassle, and many costly mistakes that could potentially jeopardize your entire endeavor. It is an important reference for the shelf of any software architect. . . . you will find this book is a complete Users' Guide that explains not only the '4 + 1 views' from the Rational Unified Process, but other approaches as well.

3 Roles and Responsibilities

This section deals with the roles and responsibilities of the IT architect. It looks forwards to a future class that will examine the “Business–IT Gap” and how IT architects can help to bridge it. We will explore the purpose of the roles and how the IT architect fits into a project or a permanent organization.

IT architect is the principal technical leader in his or her given domain. A chief architect for a major commercial, industrial, or governmental concern will be the person who oversees the technical strategy of the whole organization; such a chief architect will generally report to a Chief Technology Officer, or to a Chief Information Officer. Unlike the CTO or the CIO, who normally have some executive responsibility in a large organization, the chief architect will be unlikely to manage a large department, but will spend his time consulting and advising on how to get the best out of the organization’s investments in IT. His or her prime responsibility is to understand the technical issues and to communicate them clearly to the executive decision makers.

The domain of the more ordinary architect may not be so broad as the technical strategy of an entire organization, but the main role is the same: on a development and implementation project the IT architect is still the principal technical leader; the person who understands the whole system from the business objectives down to the coding techniques used in a program module. Most importantly, the architect is also a communicator. There is no point understanding the whole system (apart perhaps from the personal satisfaction of the train spotter) without also explaining it to all those involved as appropriate. Crucially this probably means explaining a single, coherent system in a number of quite different, but consistent ways.

3.1 The architect as technical pre-sales support

The large IT vendors and services companies are important employers of IT architects, and it is likely that most people in the IT industry whose job title includes ‘technical’ and ‘architect’ are actually working in customer-facing roles in large sales organizations. We will mention later their role in developing responses to bids and designing solutions as part of an effort to win a big development contract, but in many large organizations it has become essential to support the sales force with sufficient technical staff who can understand how complex products are supposed to work together, and can explain to potential customers how they might benefit from buying them.

In the early days of computing, most vendors employed staff to help to physically install the equipment they supplied. These members of staff were typically called ‘systems engineers’. As IT systems developed the systems engineer role specialized into hardware and software engineers. As IT became pervasive in an organization the role needed further specializations: experts at getting different products to work together, and technical staff that understood not only the vendors products but the problems of their customer’s industry or commercial sector and could advise on how best to solve the customer’s business problems with the technology. This latter role is still an important one for IT architects. Sometimes the role focuses on a specific industry sector, sometime on a range of products. Sometimes the focus is at the business application level, sometimes at the level of the infrastructure (because just running the IT has become an industry in itself).

3.2 The architect in the software development life cycle

In many software development projects it is unusual to have a full time architect on the immediate management team, although many of the large consulting firms employ one of their more technical and senior developers to do many of the architect's jobs. Typically this individual will also be the team leader of a small infrastructure team that acquires and manages the development and test infrastructure, and works closely with the customer's IT staff to hand over the finished article at the end of the project.

In other projects the architect may be involved part time, perhaps as a reviewer at critical stages of the project. These interventions are typically to make sure that the finished system will work correctly. There are several aspects to this:

- Traceability — it is important in many industries, especially tightly regulated ones like financial services, for the organization to be able to demonstrate that they are making good use of their investments. For an IT project, this generally means being able to demonstrate (to an auditor) that a given set of investments in software or hardware actually supports some business function. The financial planners like to be able to see that the money spent is providing some tangible benefit. The architect is the professional who is most able to explain how what is actually implemented supports what was originally required.
- Performance — a familiarity with what is physically possible in computing and a sense of confidence in simple numerical analysis are vital skills for an architect. Making sure that the system will perform adequately is second only to making sure that it does what was required. In fact adequate performance is always (at least) implicit in every requirement statement. One of the roles of the architect is to tease out these implicit statements and to ensure that the numbers are captured along with the requirements.

  The numbers will include: likely numbers of users; seasonal, weekly, and daily patterns of usage; likely numbers of transactions of a given type; the worst response time for a given transaction that would be acceptable to the business users; the rough size of data required; the geographic distribution of the users and the computer locations; the likely network capacity and response times available.

  As the design of a system progresses, these numbers will be built up into a performance model which will be used to predict the performance of the whole system, to design the performance testing, and ensure that improvement efforts are concentrated on the right places.
- Reliability — closely related to the performance model is the reliability or availability model. The architect needs to take a broad view of the components and context of the whole system. This view will help to inform decisions about which components are worth duplicating, how back up should be managed, where to rely on a third party system for input, and where to remain self-contained and independent.
- Cost — the fourth major area that an architect will need to review on all commercial software projects is the cost of the system. Designing to a price involved compromises and trade-offs between function, speed, and realistic project time tables. The architect is a crucial player in making these decisions on a project, as cutting costs here and now may have an even more expensive effect over there and

then. It is vital therefore that the architect should have a good grasp of the fundamental economics of computing and the basic principles of financial analysis.

Besides the management and reviewing role outlined above, the architect working on software development projects will also be responsible for design and development standards (and overseeing their enforcement), development environments and tools, and writing design guidelines and rules. As ever, such rules and guidelines are useless if no-one knows about them or follows them, so the architect must get involved in a communications programme and must ensure that the management and compensation arrangements work to encourage staff to follow the rules.

3.3 The architect on an end-to-end implementation project

As noted above, at IBM the IT architect job evolved from one that used to be called system engineering. This job still exists, and from time to time architects are still required to get involved at this end. Many of them rather enjoy it. It is the world of product specialists and technical support staff who have deep knowledge in narrow specialized areas; often the architect's ability to grasp the big picture makes them very good at managing all the technical details without getting 'lost in the weeds'. Nevertheless it is important for an architect to know when to seek specialist advice.

At this level the job consists of making sure all the infrastructure works. This infrastructure clearly includes all the hardware and communications equipment, and also large parts of the software 'stack' that runs it. But is also includes some softer aspects: these include system management and operational planning.

Hardware considerations

At the lowest level the architect-as-system-engineer must ensure that the necessary equipment is in place in time for the planned system, or for the planned changes. This means working closely with the vendors of the kit to ensure that it is ordered in good time (long lead times are still with us for large IT systems), that the machine room will have enough space to accommodate it, that the power and air-conditioning will cope with the extra load, that the operator manuals will arrive on time, that the doors will be big enough, and so on. This is known as operational readiness, and the more organized vendors will do much of this for you: they often have their own planners and specialists who will provide the necessary checklists, and environmental planning information.

The architect is also responsible for making sure that the *right* kit is ordered and that there is enough of it. As early as possible in any design effort, the architect (or a delegated specialist in the architecture team) should start collecting numbers of users and transaction types from the business requirements, and begin to translate these numbers into physical processing requirements, and network capacities. In RUP terms this is a combination of the process view and the deployment view.

Even if the hardware issues are delegated to a specialist, the architect must ensure that the specialist has a good understanding of the whole system and the overall purpose, so that he or she directs their efforts to the right ends, and can communicate what is required

to the appropriate vendor. One of the key skills here is to be able to judge from the results of benchmark tests or the vendor's planning information how the planned business transaction workload is going perform on a given machine. In extreme cases, or for radically new workloads, it may be necessary to get the vendor to prove the performance of the system by running pilot or prototype tests as part of the development cycle.

3.4 Managing the software stack

Software takes over where hardware leaves off, and in some cases slightly before. It is not unknown for the 'microcode' in some IT systems to cause compatibility problems, even for kit from the same vendor. As a system engineer the architect must make sure that all the software required to develop and to run the system is in place at the right time, and, in view of the cost of monthly software licence rental on large systems, not a moment too soon.

The software concerned includes the basic operating systems for all the hardware, middleware, and development tools.

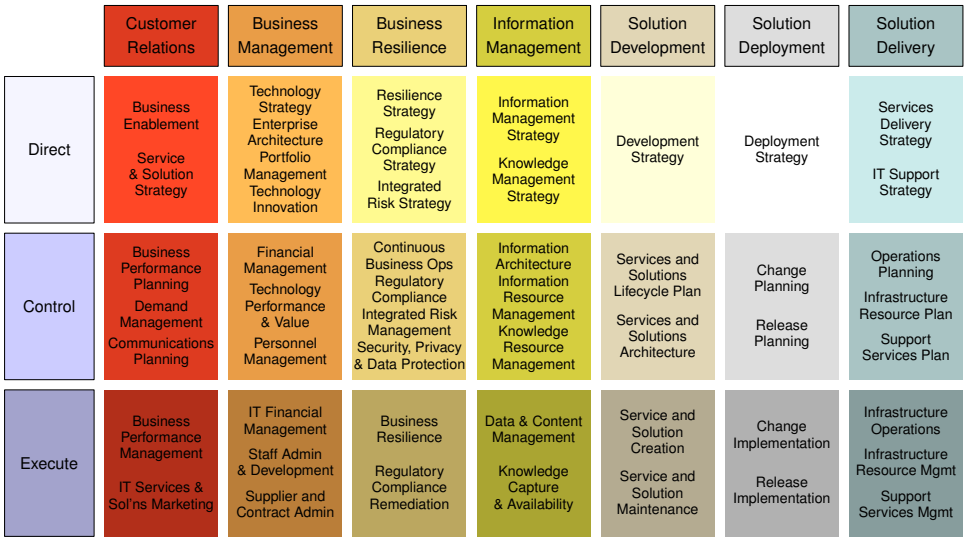
- The operating system can be surprisingly complicated. Each machine type has its own operating system, and a vendor will typically support several different versions or levels of each system. The architect must understand all the differences and potential incompatibilities and ensure that necessary upgrades are planned into the project schedule and budget.

  With Windows — particularly Windows Server — there is the additional worry about security patches. Other operating systems are not immune from security problems, it is just that Windows has many more than any other system. So many that a special process for finding and applying patches is necessary for any system that makes extensive use of Windows servers or clients. The main trouble is that applying patches takes time and effort on a regular basis.
- Above the operating system is the middleware — a wonderfully elastic term that includes everything other than the operating system that is not actually part of the system you are building. The choice of middleware has a profound influence on the design, development, and operation of a system so it is important to get it right. The basic purpose of the middleware is to provide common services to all the components of your system to allow them to communicate efficiently and to make your developers more productive. Crucially the choice of middleware product will normally be made at the enterprise level so on an end-to-end engagement you will probably need to use what it already in place. The architect's job is to assess how suitable any existing middleware might be and to ensure that it is used effectively.
- Strictly part of the middleware, but so important that we should mention them separately, are the database management system and the transaction management system. These are both areas where deep specialists make entire careers out of one special aspect of a product, and where the architect will need assistance with the detailed design issues. It is also an area where the choice of product will normally be one of the 'givens' at the start of the design. The end-to-end architect must know enough about the characteristics of the chosen systems, particularly the performance characteristics, in order to oversee the detailed design effectively.
- All of the above items in the software stack will help to determine the choice of software development tools. Different people prefer different tools, and often form

strong sentimental or religious attachments to them, so the architect's job of choosing a development environment and policing it is often not a happy one.

Beyond the hardware and the software stack, the architect-as-engineer must also think about the management of what is being built both during development and once it is released into production. The disciplines of system management are well developed and widely adopted in mature IT organizations. There is a (rather dull but worthy) set of standards called ITIL (the IT Infrastructure Library) that describes all the disciplines involved from change management and capacity planning to instrumentation and provision of a help desk.

IBM has developed an architectural view of these disciplines and their associated processes known by the snappy title of the Component Business Model for the Business of IT (CBM4BoIT). This divides the job into three layers — direction, control, and execution — and several functional divisions. This is illustrated below.



The architect needs to be aware of what is going on at all levels in this model, but will spend most of his time working in the upper level (direct), setting standards and policy. Time spent working at the execute and control levels however provide invaluable experience and insight into the job of managing communications and information systems.

3.5 The architect as town planner

If the architect working on a single system implementation project is like the master mason working on a medieval cathedral, overseeing the whole conception and making a link between the sources of funding and the actual construction work, then the IT architect working across several projects is more like a modern town planner.

The idea of the architect as planner is written up extensively in Roy Schulte's 1997 paper called 'Architecture and Planning for Modern Application Styles' [3]. Whereas the traditional architect will be concerned with the design of one building at a time, the planner

is concerned with the overall development of the whole town, and whether each proposed new building will fit into and enhance the town's overall environment.

The planner will do two other important things: he or she will be responsible for determining what common services (like water, sewerage, gas, electricity, etc) the town should provide to each individual building, and how much to charge for them; and he or she will also set out guidelines for building design and construction and will enforce a governance regime to ensure that all new buildings comply with the rules.

There are direct parallels to these activities in the IT world, and the IT architect as planner — or the 'enterprise architect' as he or she may be known — justifies his or her existence by the efficiency of the overall IT investment and the value it delivers to the enterprise. This is often a surprisingly hard thing to do. On paper it may seem obvious that one organization does not need two or more personnel management systems, but historical issues (for example the merger history of the organization) and political considerations (department A will not work well with department B) frequently complicate the technical decisions that an architect must make. In the interests of keeping projects small and manageable it may even be more efficient to allow a degree of duplication and overlap, provided it is part of a larger design. In some cases the architect may even want to encourage it. Provided the interfaces are clean and the different problem domains are well understood it is possible that the organization will benefit from having two competing solutions to the same problem. This is much more likely to be true of transactional systems than master database systems. It may be a good idea for a phone company to have several competing billing systems that can be used on different batches of call data records, but it is rarely a good idea for one company to have more than one master reference database. It is up to the enterprise architects to decide what is appropriate in each case. This frequently involves some interaction with the financial managers of the enterprise, or at least sufficient awareness of the business processes.

Similarly the enterprise architect should be aware of financial parameters under which a company operates and should ensure that individual designs conform to any cost constraints. The architect Arne Jacobsen was well-known as a perfectionist: everything in his buildings had to be just right. He designed and specified everything down to the furniture, the door handles, and even the shape of the bricks in some buildings. The peak of Jacobsen's working life was in the post war boom of the 1950s and 1960s. He died in 1971 just before the first major oil price crisis. It is perhaps unlikely that a Jacobsen could have survived in the modern economic climate. Today architects — in information technology and in real construction — make use of standard components as much as possible in order to reduce cost and to promote interoperability.

3.6 The design authority

Experience gleaned from many communication and information system development and delivery projects, shows that projects need more than just good project management in order to succeed. They need technical **governance** and they need IT architects working in a **design authority** to deliver it.

Projects rarely fail on unsolvable technical issues, but have often failed or faltered on poor IT governance.

- The requirements may be incomplete, inconsistent, or incoherent (or all three), and it is normally too late (or at least expensive) to fix them after a contract has been signed.
- The planned hardware and software may not be big enough to cope with the workload; or the actual workload may be much bigger than the business anticipated.
- There may be other technical risks that were not foreseen. Worse still there may be technical risks that *were* foreseen but no plan was made to contain them. 'We didn't design it to cope with circumstances that bad. . . '
- The parts of the solution may not be fit together when it is (finally) tested as a whole. 'We thought they were doing that part. . . '
- Performance issues may not be picked up until solution is live, whereas the team should have focussed on performance throughout design, build, and test.
- The availability of legacy systems may suffer because of the introduction of new systems. 'We didn't realise that system interfaced to those servers as well'.

It is the core job of the IT architect to provide good IT governance, but he or she cannot act as a one-man-band. The architecture and design principles and standards must be owned and maintained within a single central body. This helps to ensure ongoing solution integrity over time (and when key individuals move on), and to ensure that different projects adhere to the business and IT strategies of the organisation. A central design authority can also ensure that the organization makes the most of its IT investments by encouraging reuse of whole or partial solutions.

At its simplest, the design authority exists for two reasons: (a) to carry responsibility upwards (also known as 'technical accountability') and to exercise authority downwards ('technical control').

The people who control the project need to have the assurance that the solution will work. Since perfect knowledge is usually unachievable, this resolves into a tradeoff between spending money (on designing out all the theoretical problems, on buying skilled people, on frequent reviews, and on spending longer time on all this) versus accepting the risk that the solution might not work and will then require a lot of money be spent on fixing the real problems. An investment in a design authority is a sensible use of money to reduce the risk that the solution will fail.

Once the solution is designed on paper, the build teams will need to have guidance in some areas on how to build it, In particular on how the overall solution is split into smaller elements to facilitate coding and testing, and on what is the best sequence for building so that tests can be done in a structured manner. The design authority is also in the best position to see (and to explain) the technical dependencies on external or parallel activities.

The design authority team will normally be led by a chief architect, and will be made up of several specialist architects doing one or more of these roles:

- Application architect — often the second most senior architect, responsible for the overall shape of the solution or the enterprise plan. Understands and can explain how each major subsystem fits together and what each one is supposed to do.

Designs the interfaces between different applications and parts of applications, and governs their use. Decides on when to buy or build new applications, when to enhance or replace existing ones. Defines the overall portfolio of business services that the enterprise needs.

- Infrastructure architect — in charge of making sure that the right equipment is in the right place at the right time, and that all the middleware is specified and works properly. Works closely with the performance architect to do the capacity planning and hardware sizing. Works closely with the application architect and the system management architect to understand the availability requirements and to design appropriate redundancy. Responsible for the operational model that maps functional components and services to the servers and systems that actually implement them.
- Data architect — owns the enterprise data model and the inter-application messaging design. Responsible for making sure that the applications use the data base effectively. Responsible for planning for the effective use and exploitation of information. This may include the design and supervision of an operational data store (working copies of live data) or a data warehouse (extracts from live databases, summarized and enhanced to help the business management understand the state of the enterprise).
- Integration architect — owns the middleware design and ensures that all applications make effective use of it. Works closely with the data architect on the messaging design and the application architect on the application design.
- Security architect — responsible for the various aspects of system security: principally user access control, user authentication, data integrity, auditing, network intrusion prevention and detection, use of data encryption or anonymization technologies, fraud detection and prevention, and overall system integrity (which essentially means ensuring that the system does what it is supposed to and nothing else).
- Performance architect — owns the performance model of the system, works closely with the application architect to build the users' use case model of the system, and then to derive a corresponding transactional model. Works with the development team to ensure that appropriate instrumentation is added to new applications. Works with the infrastructure architect to ensure that the component model is sized effectively to support the planned transaction and user volumes. Works with all the others to ensure that the system as a whole will provide adequate performance for a reasonable price. Works with the development teams to design, plan, and interpret the results from performance testing.
- System management architect — responsible for the design of the operational support systems including availability and performance reporting, resiliency planning, change planning, release management, operational procedures, data back up and recovery, metering and billing, design and planning for help desk and other support resources.

The combined design authority team works in four related areas:

- Requirements management: in an ongoing project, or for an enterprise on a continual basis, this consists of ensuring that the requirements for the business are

well understood by the development staff and working with the business to clear up any misunderstandings (on both 'sides'); understanding how the requirements evolve and change over time, and managing those changes through into real systems. In order to do this the team will establish a process to allow them to trace each implemented function back to a particular requirement. It is important for the business to know that it is making sensible IT investments and to be able to see that the money spent on system X is supporting this valuable business requirement Y. The increased level of corporate scrutiny from external industry bodies (as part of Sarbannes-Oxley and other legislation) means that the design team have a vital role to play in not only helping to ensure that money is spent in the right places, but also in demonstrating this to interested outside parties.

- Enterprise architecture: setting out the overall technical policy and direction; making sure the divisions between different projects or sub-projects are clear and widely-respected; establishing and promulgating a consistent set of design methods and tools; managing the high-level interactions of successive change programs into a cohesive whole; avoiding unpleasant or expensive missed dependencies; and ensuring that the interfaces between different cooperating parts work well.
- Solution assurance: helping to develop new proposals, overseeing project plans (especially to coordinate changes at critical times); signing off project 'base line' plans and budgets; running reviews at regular milestones in individual projects; making remedial interventions as needed.
- Design management: oversight of the design, development, and testing processes; being the final 'court of appeal' for design issues and maintaining a set of 'Architectural Decisions' of global importance, that sets out each decision, the arguments, and the rationale for the chosen option (well maintained this can save an immense amount of futile argument over the same design choices in different projects); responsibility for the containment of technical risk and making the financial management team aware of the implications of the risks; control of the change management processes across all work.

There are plenty of examples of well run projects that have had good design authority teams; unfortunately there are rather more examples of projects that have not been able or willing to set up a design authority team. Many of these projects struggle to contain technical risks and issues and many of them fail to deliver all or part of their objectives.

In successful teams there is always a close working relationship with the business leadership. Sometimes there will be a parallel business design authority team, but if this is the case, some single person needs to be in direct control of both the technical and business teams.

3.7 Consulting engagements

Senior IT architects, especially those working for large IT services companies, often work as consultants, advising clients about how to set up and run their own IT architecture departments, or about a wide range of other aspects of communications and information systems strategy.

This work might include reviews of the effectiveness of a client's IT organizations, or reviews of particular projects. It might include research into different aspects of technology

futures or best practice. It might include a study of a range of options for change where an independent voice and a fresh point of view is valued.

The senior management in many organizations, particularly in the public sector, either cannot afford to keep their own senior technical staff or don't trust them. In these circumstances an external IT architect may be called in to evaluate a position, to recommend some course of action, or to work as an agent for change. They may even be seconded to the client organization for a period of months or years. These assignments require a special mix of skills, experience, tact, and an ability to influence other people without direct managerial control. It is vital to be able to establish one's credibility and usefulness with client staff at all levels, and to be sensitive to the local issues and political pressures. The work can be very rewarding for the right individuals as it requires not only technical experience but also personal skills to make a positive difference at a client. These assignments are often done by one architect working alone, or perhaps a small team.

Senior IT architects are also often members of a professional body such as the British Computer Society or the Institute of Electrical Engineers (IEE), or of a cross-industry standards body or users' group. These forums provide an important means to maintain technical vitality for individuals and organizations, and provide opportunities for development and career advancement. They are also often the forum for senior architects to 'give back' some of their experience and insight to the rest of the community. In the large services organizations this experience is usually distilled as a formal **method**. The services part of IBM has a very large method called IBM Global Services Method. The formal architectural deliverables and techniques are introduced below (page 38); the senior architect's contributions are normally in the form of short, practical **technique papers**.

3.8 Working with other IT professionals

The author Fred Brooks best known for his book *The Mythical Man Month* [9] is a rich source of anecdotes and quotes. On the book itself Brooks once said that it's called the bible of software engineering because everyone knows they should read it and act upon what it says, but nobody ever quite manages to do so. On IT architects he maintains that they should be "protected from managers and *managing*". This section examines how IT architects work with other IT professionals, both managers, peers, and staff.

Despite Fred Brooks' dictum it necessary for IT architects to deal with managers, especially in a large bureaucratic firm, and also to be managers from time to time. A good technical manager will promote the career of an IT architect, look out for suitable opportunities, and allow time for education and personal development when setting objectives. It is also incumbent on the IT architect to promote their own career path; setting a clear technical direction often helps ('I want to be a performance architect' or 'I want to be a data architect'); make sure that your manager knows what you want to do, and that he or she gets good feedback about you on jobs that are in your preferred direction.

In some organizations or on a particular project your reporting manager may also be your task manager; if he or she is a IT architect then you should be looking to learn from them, and to divide up the work in a way that best suits your combined talents. If you are working for a non-technical manager, then you will need to think about your work slightly differently.

Most good project managers have a clear idea about what the technical architecture team are supposed to do and will have a set of expectations that you will need to meet and to manage. Clearly you will have to judge each project situation on its individual characteristics, but in general it is a good idea to get ahead with a plan for your work, be prompt in reporting your progress to the project leader, and to get yourself (or one of your team) established as a key reviewer at any review or for approval of significant design milestones.

The important things are that you are left to exercise technical control (as defined above in the section on the design authority), and that the project manager relies on you for input on technical items such as task sequencing, interlocks, issues and risks.

Providing this input is one of your key tasks with a non-technical manager. You must make sure that they understand enough about the technical problems in order to make informed decisions about the timing of the project and the allocation of resources. The ability to explain what is important about a technical issue is one of the key attributes of a really good IT architect. The important thing is to concentrate on the important aspects of the particular issue and not to get bogged down in the fascinating technical details; it often helps to imagine what the other person will be thinking and what is important to them. This applies equally to other non-technical decision makers that you need to influence and persuade.

The second major class of colleagues that an IT architect will work with are the technical specialists. There is a wide range of people involved in specialist technical work, not all of it to do with communications and information systems. The specialists will include marketing experts, business analysts, strategy and change consultants, industry specialists, hardware specialists, and software product specialists. At their worst they will have a narrow view of the world, blinkered as they are by the need to understand their own field completely. Your job is to make sure that you get the right help out of them, and that you explain enough of the ‘big picture’ context to them so that they can contribute effectively.

3.9 The architect in procurement

Procurement is the name of the process by which large organizations acquire communications and information systems (or any other major capital outlay but it is only the IT aspects that concern us here). Most IT architects spend a significant proportion of their time involved in either developing procurement documents or responding to them. This is not because it is easier to theorize about a new system than to concentrate on running an existing one, but because the best time to apply the architect’s skills to a system design is right at the front.

Work on bids can be done on either side of the fence between suppliers and customers. For a customer the architect must work with the business leaders to develop an effective set of documents to express the requirements in terms that can be understood properly by the bidding vendors and which present the work as an attractive proposition. These documents are generally in a formal series. The first is a ‘request for information’ (RFI) that outlines the requirements, and asks for short responses from vendors to outline what their products can do and how they would go about enhancing them or building a solution for you based on them. The responses to the RFI are used to draw up a short list of serious

contenders. The formal document response is normally supplemented by private conversations, presentations, and briefings. The customer IT architect would be expected to attend to understand the technical issues involved and to ask questions that reveal strengths and weaknesses.

The requirements may be re-evaluated based on the initial vendor responses about what is possible and how to approach the problem. The next document is a more formal 'request for price quotation' or RFP, which sets out some business objectives, an outline specification for how to solve the problem, and requests a detailed response from the vendor showing how *they* will solve the problem, and how much they will charge. This document is sometimes also called an 'invitation to tender' or ITT. In the public sector, which is subject to European procurement rules, the normal form is for this document to take the form of an exhaustive 'output-based specification' (OBS) which states in great detail *what* the proposed system should do, but (in theory) leaves it up to the vendor to show *how* their system will do it. The IT architects in the customer and in the vendors will be the primary authors of these documents and their responses.

On the other hand, except in a very few cases, they will not be the final evaluators or decision makers on either side. This can lead to much creative tension during the bid cycle. The technical team may be convinced that a solution can be done for so much, only to find that the commercial team have sold it for 30% less. A close working relationship between the technical and commercial teams is essential to success. This means that the IT architects need to understand enough about the commercial aspects to know what solutions are appropriate, and to establish a good rapport with the non-technical staff, so that their judgement is trusted.

3.10 Skills required for architecture roles

A common theme running through this section has been the need for the architect to have good communications skills — both oral and written — and to be able to explain complex issues in simple, appropriate terms. They must also be able to build good relationships with others in order to gain their trust, and to understand enough outside their own domain to be able to make good technical judgements. For those that associate technical architects with the narrower sense of architecture as the nuts-and-bolts of the solution, this may be a surprisingly wide set of skills.

4 The common language of architecture

In this section we discuss some of the terms widely used in architectural discussions. You will have had your fill of them from the pre-course reading [3, 4], and this is your opportunity to get them clear. In the class room you should be sure to discuss terms that you would like to understand better.

The ideas that the common vocabulary represent form a set of patterns that you can reuse in designs and problem solving. They are a great aid to productivity, saving you from re-inventing everything from scratch each time; but don't rely on them too much. If you do you might stifle your creativity and your ability to come up with new solutions to existing

problems. For a general introduction to this way of thinking you should try Martin Fowler's books *Analysis Patterns* [10] and *Patterns of Enterprise Architecture* [11].

4.1 Open systems

In the early days of computing all the vendors designed and built their own systems with their own operating systems and their own application software. Once you had chosen a vendor, you relied on them for more or less everything. If you wanted to move vendors you have to start all over again. If you worked in an organization dominated by powerful, non-technical but budget-holding groups of users, (like an investment bank), you would acquire over time communications and information systems from a wide variety of different vendors. All of these systems would be completely incompatible.

Then in the early 1970s two American systems programmers had a clever idea. They had been working on a new operating system, but they had their departmental budget cut and lost the machine that they had been working on. Forced to migrate their work to a new machine architecture, they decided to rewrite most of it in a high level programming language (called 'C', because it was a successor to 'B') and use the existing compiler for the new machine. The operating system became Unix, and it sparked a revolution in price-performance across the IT industry, as Unix could be (and was) ported to any machine with a C-compiler.

This meant that you did not have to rely on the hardware vendor for the operating system but you could stick with Unix. It also enabled the development of a whole new segment of the IT industry, the major application package vendors. Companies like SAP now only had to write their software for Unix to have it available on a dozen or more hardware platforms. This was called **portability**, and a cross-industry effort was started to set out standards for developing portable applications. Unfortunately the progress of the standards-setting groups was slow, and meanwhile IBM went on making non-open mainframes (as well as other systems) and a company called Intel produced the 8086 microcomputer.

Once Intel hardware running Microsoft Windows had emerged as the only choice for small scale computing, the Gartner Group were able to write (in 1997) this about open systems.

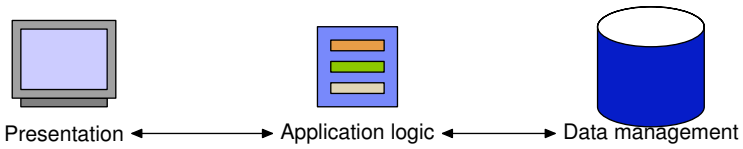
> Open systems are based on the idealistic premise that portability and interoperability can be achieved across many different base products as long as they all conform to the same multivendor interface and protocol specifications. However, in many areas of IT, this premise has not been fulfilled in practice because the standards have been slow to develop and are incomplete, and vendors have continued to deviate from a strict implementation of the standards. Interest in open systems peaked in about 1993, and most clients, even those that were previously the most enthusiastic in their pursuit of this strategy, have since reduced their efforts and their expectations in this direction. Most clients continue to pay attention to the importance of *de facto* standards, but many now place relatively little reliance on official standards unless they also happen to be (or appear likely to become) *de facto* standards.

Since then the pendulum has swung back quite a long way towards the idealism of the 1980s. For one thing many of the standards about open systems are now quite mature and

surprisingly usable. For another the rules of open systems have moved up a level or two. Today support for open standards is less about having software that can run on many different platforms (although Java and many other interpreted languages coupled with increasing hardware power have solved most of that problem) and much more about **interoperability** between systems across a network. Today it is perfectly possible, or even normal, for a distributed application to use a platform-neutral browser as the front end, a Windows server as the web server that serves dynamically generated HTML pages, and a Unix or mainframe server in the background to run the database. All the interfaces (well most of them) between them will be defined according to a more or less formal interoperability standard.

4.2 Parts of an application

In describing application systems and thinking about what architecture is suitable for a given problem, it is convenient to split a given business application into three elements.



Presentation logic The management of the user interface: personal authentication, presentation of information, capturing input, printing, controlling pointing devices, managing displays, speech recognition, speech synthesis. Design in this area is a major academic discipline involving psychology and industrial design skills. There are many good books on the subject, but you might like two that give a slightly broader view on the subject. *The Design of Everyday Things* by Donald Norman and *User-Centered Design of Systems* by Noyes & Baber.

Application logic That part of the application that processes data captured from the other parts. This is where the business rules of an organization are implemented. The part of a billing system that works out the discounts and the product hierarchies, or the part of a treasury management system that does the foreign exchange risk-spreading calculations. Also known as the number crunching. In the days when the only input device was a card punch and the only output was a line printer and there were no databases, this part was all there was.

Data management logic The database subsystem, which maintains the persistent store of information for an application, serializes multiple accesses to the same data by many users, maintains the integrity and consistency of the data in the system. Since any database also involves some amount of programming to make it work — typically done in SQL — you will also see the term **data access logic** here. The term is supposed to cover **definition of data** using SQL as a data definition language (DDL) and **manipulation of data** using SQL as the data manipulation language (DML). To manipulate data means to read, write or delete it.

This Caesarean division-in-three-parts provides a convenient model and in some ways is an ideal way to organize a program, but in many real world systems, the division is not obvious. A database may contain many stored procedures that implement business logic; a

Windows application may embed business rules in the presentation and capture of the data from the user. However this three-layer model is universally understood and helps us to discuss and design client/server systems.

4.3 Clients and servers

In the strict software design sense, a server is a program that provides a service, and a client is a program that uses that service. A system composed of clients and servers is generally known as a client/server system. It's really that simple.

You can implement such a system on any hardware platform you like, it is a good way of organizing most systems. It is how many modern large mainframe systems, such as the one that runs your bank's cash machine service, are designed. However in many cases the server part of the system runs on a centrally managed, fairly large, multi-user platform, while the client parts run on workstations or PCs. This has led to the confusing, but widespread, usage of "client" as a synonym for PC, and "server" to refer to any departmental or larger computer (of any type). Following the strict sense above, you will note that the same system can be both a client and a server at the same time.

Could you describe how to implement a client/server system from scratch? If so you may get called on to do so in class. Here is a handy crib. A client/server system needs two pieces of matching code. The server-piece will look a little like this:

```
import java.net.*;
public class Server {
        public static void main(String[] args) {
                ServerSocket listener = new ServerSocket(4444);
                Socket connection = listener.accept();
                /* read from and write to connection */
                connection.close();
                listener.close();
        }
}
```

This is a very minimal Java class implementing the fundamental client/server pattern. The server sets up a listener and then waits for connections. Note that accept() will block until a client connects to it.

The minimal client code might look like this:

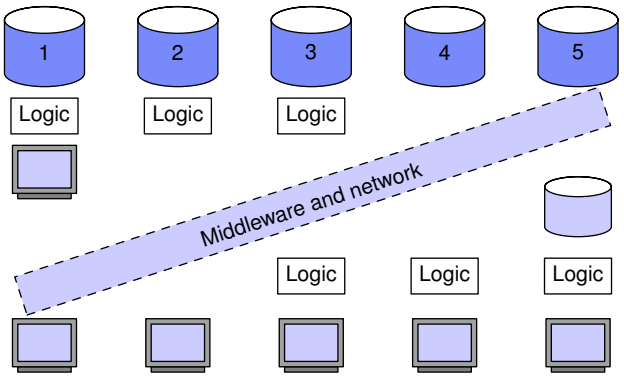
(For a proper introduction try the online networking tutorial at Sun's `java.sun.com` site).

```
import java.net.*;
public class Client {
```

```
        public static void main (String[] args) {
                Socket connection = new Socket("some_host", 4444);
                /* read from and write to connection */
                connection.close();
        }
}
```

4.4 Distribution architecture

If we draw a picture of the three layers discussed above, copy it five times from left to right, and then draw a line diagonally from bottom-left to top-right through all five stacks of layers, we get an approximation to another Gartner group model: The five types of distribution architecture.



You need to imagine that above the diagonal division, everything is running on a central machine — a server if you like — and below it everything is running on several, distributed machines — probably PCs, but we can call them clients. For fairly obvious reasons the five types of architecture are known as

1. distributed presentation
2. remote presentation
3. distributed logic (they mean “distributed application logic”)
4. remote data management
5. distributed data management

In the middle, connecting the upper and lower parts we have middleware and network software. All of these architectures are ‘two tier’ architectures. Over on the left we have **thin client** solutions, over on the right **fat client**.

☞ *Take a moment to characterize some systems you have worked with in terms of this model.*

This neat, symmetrical view of the world is much too simple of course, and you may have found that you have worked on systems that did not fit into any of the boxes very well. The most glaring simplification is that in the real world there is often more than one horizontal division. Consider a typical SAP system: the SAP client software runs on a Windows PC and talks to the application server that might run on a big Windows PC or more often a

Unix system, meanwhile the database is running on a third platform, perhaps a mainframe system. This is called a ‘three tier’ system, and was the dominant model for Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM) packages throughout the 1990s (and is still widely used).

The reach of the Internet has encouraged some package vendors to take this even further and to provide another layer that interfaces to the application server layer, and represents information to Web based or other remote presentation devices. This would be ‘four tier’, although the designers often get carried away and go for the term ‘n-tier’ when they get this far.

4.5 Object-oriented terms

The concept of the system as ‘black box’ is an important one in the architecture of complex systems. As we discovered in the ‘what happens next’ exercise we deal with complexity by hiding it behind interfaces: we don’t have to program to understand the electrical signals from the mouse, we merely capture events provided by the operating system that represent mouse movements and clicks.

In object-oriented programming systems (OOPS) this is called **encapsulation** and it refers to the definition of an object by its interfaces. More than that encapsulation also protects the data within an object from unexpected change. The idea is that if we can break a programming task down into small pieces with well defined inputs and output, then we can solve it more quickly and our solution will be more robust. The same idea works equally well in large-scale system designs. However you should note that the use of encapsulation implies that your design is conceived in terms of objects that exist both as an ideal definition of a business ‘thing’ and as instances of the ideal definition. For more details of this idea, see Plato *The Republic*. In practical terms this means that your design will allow for the passing of messages from one instance of an object to another. Other OOPS terms, such as ‘inheritance’ or ‘polymorphism’ do not obviously scale up in the same way.

4.6 Transaction processing

An important subclass of applications are known as transaction processing systems. In these system each interaction, or related set of interactions, with a user can be defined as a transaction: from the system point of view either the whole transaction should complete or none of it. The most obvious example is the withdrawal of money from your bank account using a cash machine. There are several steps to such a transaction:

1. Insert card
2. Key in your password or number
3. Select withdraw cash option
4. Enter amount to withdraw
5. Check available funds in account
6. Return card
7. Update account balance
8. Dispense cash

Not only is the order of these steps important (some machines dispense cash before they return the card — not a very good design), but it is important that they are all completed. You would not feel very happy if the system updated your account, but then did not give you the cash.

Note that this example implies two parties to the transaction: there are always at least two (*can anyone give a counter example of a transaction with only one party?*), and often more in complex transactions. Transaction processing systems manage all the parties to a transaction and make sure that all the steps complete. Another way of thinking about this is that they make sure that any partial changes are removed if the whole series of steps fails to complete, for any reason.

To help understand the nature of transaction processing designs, the jargon is that transactions have ACID properties. This is not a reference to hallucinogenics, but yet another acronym. It stands for atomic, consistent, isolated, and durable, as follows:

- **Atomic:** Each transaction executes completely or not at all—and leaves no partial results. This is true even after a system crash and recovery (part of the durability requirement). It also is true for concurrent transactions.
- **Consistent:** Each completed transaction moves the system from one consistent state to another. For example, consider a financial system with a consistency requirement that at all times assets must equal liabilities plus equity. Executing a transaction that records the purchase of a computer using a purchase order would violate the consistency requirement if only the increase in assets was recorded. To guarantee consistency of the database, the corresponding increase in liabilities must be recorded. If the liabilities cannot be increased for any reason then the increase in assets 'rolled back'.
- **Isolated:** Each transaction executes as if it were running alone, even though transactions may run concurrently to improve performance. For example, if a transaction transfers money from one account to another, a different transaction is not allowed to view an intermediate result, where one of the balances has changed but not the other. If a transaction were able to read such a result that was never true in fact, it might make an improper decision. It might refuse credit to the account holder, or even make a change to the data by fining the account holder for falling under a minimum balance.
- **Durable:** After a transaction is complete, the results that have been committed will not be lost because of a system or storage failure (processor or disk crash). If a system is to provide durable transactions, then as the system makes updates to its database, it must write a transaction log (to a different storage medium than the database itself). The log will allow it to undo those changes if the transaction should later fail for some reason (possibly because of a system crash), or redo the changes in case the transaction succeeded without all the updates getting saved prior to the crash.

4.7 Design style

Application designers building complex systems will naturally split them up into many component parts. Making this division requires the designer to choose an appropriate style

for dividing the application. The fundamental two choices are tight-coupling or loose-coupling; both have advantages, which is better depends on the problem that the designer is trying to solve.

When two systems are “tightly coupled” they communicate at the application and/or data level. Interactions between components are normally synchronous and real time; that is the calling component will wait for the called component to complete processing and return control. This is sometimes conceptually simpler for the designer than periodically checking to see if a result is available yet, but it can lead to large and complex systems with many dependencies. On the positive side it also means that it is impossible for one part of the system to get out of step with other parts. No special design is required to maintain synchronization.

Data flows in both directions as function parameters and return values. Calling programs must know the right parameters to pass and know what values to expect in return. Called programs must know how to signal abnormal conditions and errors.

The availability of the overall system depends on all the component systems: if any one component fails then the whole system fails. A tightly coupled system is like a chain: it is only as strong as its weakest link.

Tightly coupled systems can also be hard to enhance and maintain. Adding new components may mean updating many unrelated interfaces, and it may be hard to extract functions and routines that could be used elsewhere.

In contrast a “loosely coupled” system must have clearly define interfaces between components, and each component is typically a small self contained, reusable block of function. Components communicate with asynchronous messages. Component A sends a message (via some medium); some time later component B receives it and may reply. Each component needs a way to send messages and a loop that sits waiting for replies. However each of the components is independent, and a well-designed loosely coupled system is resilient to the failure of a single part.

The trouble with keeping each part isolated from the other is that they may need to be resynchronized from time to time, which can complicate error handling. It is also possible for one part to generate so many messages that the other parts cannot keep up and work in process is lost. Handling these classes of errors can be difficult.

The great strength however is that small, well tried components can easily be reused in other systems, and additional interfaces can often be added without affecting the whole system.

5 Applying the vocabulary of architecture

This section summarizes some of the key points made in the middle sections of the main reading material article [3]. With some simple examples, we discuss how you can apply the vocabulary that we have just covered. The aim is to help you develop your own conceptual model of how the system that you are working with hangs together: this will help you make sense of and assimilate the details in the rest of the course.

5.1 Internal application architecture

Using the diagram above of two tier topologies, we can identify some common, perhaps 'traditional' application styles in terms of the girth of the processes on the workstations, or end-user devices.

- The classic two tier, thin-client applications are the old green screen mainframe applications, still in widespread use in the banking, insurance and airline reservation industries. This is not just because these industries are old fashioned, for some applications — such as support to bank clerks or running ATMs — this is still the best way of doing things. Your data and business logic is safe in a well-managed, secure core, and your needs for presentation are well-defined, and your user interface is deliberately tightly-controlled.
- Two tier fat-client applications were what first began replacing the old mainframe applications, and they are themselves becoming dated nowadays. Typically built with an all encompassing development environment and tied to a particular database system, like PowerBuilder and Oracle. All the presentation and application logic is on the PC, while the database sits on a central system, probably connected to the PCs on a local-area network.
- Many two tier environments especially where there is a strong database vendor involved have now evolved into what Gartner like to call 'plump-client' environments, where large parts of the business logic is moved into programs stored in the database itself ('stored procedures'), leaving the client thinner, but not that thin.
- Still more two tier environments have acquired an extra middle tier. This middle tier will be running on a system similar to the one running the database, or often on the same system. To the database server it will look like another client, but it will be handling the requests for service from many PCs. There are many advantages to this design: the number of idle connections to the database is reduced, which means the database system will use less memory; the design works better where many of the PCs wanting access to the database are a long way away (why?); you can apply more computing power to solving your business problems, without either having everything on the PC or in the database as a stored procedure; it provides a convenient way to take advantage of other standard software — such as Tuxedo — which does a good job at the hard things like transaction management, messaging and communications.
- Beefing up the middle tier also enables us to use more generic client devices. There is an irony in modern n-tier application designs that as the PCs have got ever more powerful and capable, the applications need a smaller and smaller footprint on them. There are two trends at work here, both to do with the increasing *reach* of computer systems: one is the need to solve certain problems of presentation in more than one language, these techniques are known as internationalization (I18N) and localization (L10N); the second is the need to support more portable and simple user devices. The simpler you keep your client system the easier it is to take advantage of standard solutions, and the less work you have to maintain your system.

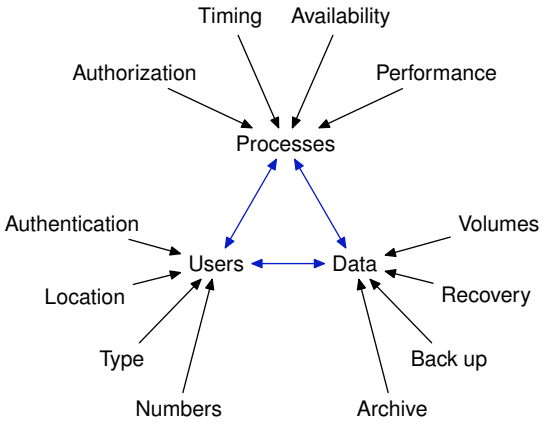
5.2 Architecture of complex systems

Moving beyond the confines of a single application package, the architects' task is to define how all the packages and existing systems should best work together. The solutions, or at

least pragmatic, working parts of them, usually involve defining the systems as components in an overall whole. The architect will need to determine the interfaces between the component parts. The problem is not confined to data in and data out, but also needs to include the practical problems of running the whole system. For each interface the architect needs to understand the volumes of data, the frequency and timing of the interfaces, the availability of them, the need for auditing and reconciliation, the need to buffer data between interfaces on different systems, the need to extract performance metrics, the performance characteristics and requirements, and so on. In addition each system needs to fit into the systems management framework of the whole installation.

Given a free hand in redeveloping parts of a system, the architect may also wish to consider which subsystems can be abstracted from the individual components and made into common shared services. Faced with the often overwhelming challenge of managing customer data across three (or more) different systems with different data models, some architects have decided to create a new, single customer data store and adapt the existing systems to use this instead of their own application-specific version.

The architect also needs to consider application topology in a physical, geographical sense; to understand the capacity of the network connections between an organization's locations; and to understand where all the users are and what they do. The task revolves around the eternal triangle of users, processes and data. On the one hand it may be best to keep a central tightly-managed master copy of key customer data, on the other hand it may be necessary to replicate this data around the world to provide the immediate access and speed of response that the users need.



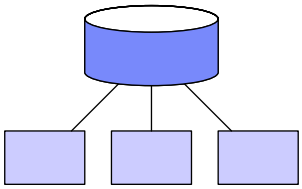
Here are a selection of topics that the technical architect needs to consider when developing an architecture for a complex system. These topics are centred on the users, processes, and data. The architect also looks at interfaces between systems and how data and messages should flow around an organization. The output of the architects work will be a definition that can be used to build the technical infrastructure. This typically encompasses everything from the physical hardware and network facilities required, through the development and system management tools, and the middleware products to be used. It can also include all the processes required to operate the infrastructure, such as performance and capacity management. Beyond this in some projects, the technical architect's may also help to determine which services should be abstracted from the

different applications and implemented as shared services.

5.3 Coordination of many applications

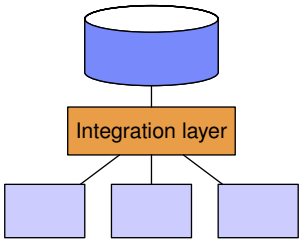
To complete this section, we now consider the main challenge for modern sites with many applications – how to coordinate the many different applications that need to communicate with each other. Whatever decisions the architect has made about division of presentation, application and data logic, and what ever considerations of topology have been discussed, the architect always has to consider the practical issues of how data from system A is going to get to system B. As you consider these different approaches, keep in mind a complex set of applications that you have worked on and think about how your architects have solved these problems, and perhaps some that they have not solved.

Direct data sharing



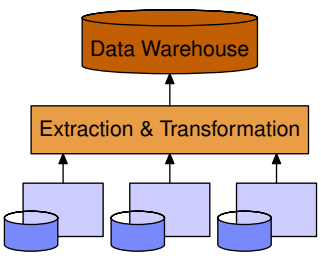
The simplest approach to the problem is to directly share a common database that holds all the necessary information. The trouble with this is that it doesn't scale well — everyone needs a connection to the same database, every time we add a customer all the different systems need to read the data to find out what was added — and worse, it becomes rapidly very inflexible — if we need to add a new attribute to our product table, then all the systems that read that data may have to change. Beyond a very few systems, with very similar data models, connected on a local network, direct sharing just does not work.

Data integration



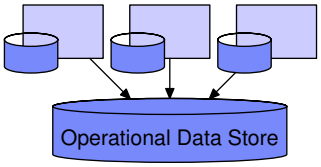
Gartner describe installations which have attempted to provide an application independent layer of software that integrates data across different data stores and applications. The idea is to have this middleware translate each data request into an appropriate format and route it to the appropriate data store. Systems like this were the forerunners of modern enterprise application integration systems. The problem — and it's still one even with modern solutions — is that it's easy to create more problems that you are solving by creating a monster in the middle.

Data warehouse



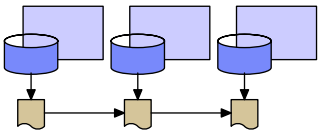
Where the communication needed between applications is not immediate, a data warehouse is a very attractive, if expensive solution. The idea is to take extracts from all the operational systems, summarize them, clean them up, reformat them into a consistent central model and then make them available to the business. The original purpose of a data warehouse is normally to support *ad-hoc* user enquiries: “how many widgets did we sell in Stockport last Thursday?”, but there is no reason why operational systems should not also use the data stored in them directly. The down side is that to make the data general purpose, you have to invest time and resources to clean it up and map it to a common data model. This is expensive to design and build and also expensive to run everyday. It also means — by definition — that the data is ‘out of date’; while this is not necessarily a bad thing, often other operational systems need up-to-the-minute feeds of data. Finally the data in a warehouse should always be read-only. This means that this approach is not suitable where mutual updates are required.

Operational data store



The ODS is occasionally taken to be another sort of data warehouse. This is an unfortunate misconception, and it is a good idea to get the differences firmly in your mind. An operational data store is meant to support some particular business *operation*, typically it will be a direct copy of production data kept up-to-date within minutes or hours. It may be gathered from several sources. It will be used for a strictly defined set of queries that directly relate to the needs of another system or some aspect of someone’s job. It will never be used for *ad hoc*, interactive queries. It will not hold historical or summarized data. An ODS can do a very effective job of coordinating data across systems in certain situations. An ODS can also make a good staging post for data that will be further processed and summarized for a data warehouse. Like a data warehouse the data in an ODS should generally be read-only.

Batch reconciliation

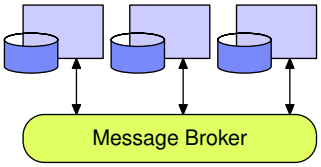


The to the problem is to take a regular extract of what has changed in your source system, and then apply those changes to your target systems. The advantages of doing this are that there is less to maintain — you can normally use standard (and therefore fast) database extract and load tools — and the networking is simple — all you do is transfer a file. If the contents of the transferred file are well understood, then it’s also straight forward to apply different filter programs to it, in order to select records for each target system, merge data from more than one source system or reformat the data to make it easier to use a standard database load tool on the target system. If the systems are running under a transaction manager you could even use the transaction logs (or ‘journals’) as the source of the changes for the target systems. In general this sort of interface is one way. It’s possible to do this in two

directions provided the frequency of updates on each side is not high and there is an easy way to reconcile changes if the same data has changed in both places.

With large numbers of systems to reconcile, the best approach is to build a central hub where all the filtering and transformation is done. This makes for more controlled implementation of the business rules and means that the feed to the target systems is more consistent. The main disadvantage of this approach is that data is not up-to-the-minute consistent across all systems.

Message broker



The high-tech solution evolved from batch reconciliation — if the only problem was speed, now we can do it immediately instead of waiting for a batch. This class of application integration system has become very popular. The general idea is that the architect designs a series of messages that encapsulate all the information that needs to flow between each application, and the sequences in which these messages will be sent — the work flow. Two 'adapters' or 'connectors' are then written for each application: one is a publishing adapter, the other a subscribing adapter. The job of the publishing adapter is to identify when an application has done something that should trigger one of the architect's messages (for example: a new customer has been added), and then create the appropriate message with all the correct data fields. The job of the subscribing adapter is to listen for messages from the message broker, and then to take the appropriate action in the target system (for example: create details for new customer). The extra layer allows the participating applications to be independent of each other, and this has proved to be very beneficial. They can be maintained on separate tracks, by separate teams. A second application in the same functional area can be added without changing the others. And so on. In other words, a message broker allows us to design and implement clean interfaces between large, complex application systems.

5.4 Service Oriented Architecture

Roy Schulte's 1997 article [3] also defines and expands on service oriented architecture. In the decade since he wrote about it SOA has become one of the trendiest approaches to organizing large systems. In some ways this is unfortunate as it has inevitably led to a wide range of interpretations of the idea of service orientation (because everyone wants to label their own peculiar approach with a popular buzzword).

To some people SOA means using web services to connect different applications together. Assuming an application provides some programmable interfaces (APIs), then web services are a way to make them available over an Internet connection. The idea is that you build an adapter for each API that can create and parse SOAP messages, that you write a description of the service using WSDL (you have to guess what these acronyms stand for), and you store these descriptions in a publicly available directory in UDDI form. When an application wants to use a particular service, it finds a UDDI directory, searches

for a suitable service, and then sets up a contract with it, and starts sending SOAP messages to it, generally expecting some similar messages in reply.

If this all sounds a little impractical or far-fetched that is because it is. The idea is a cute one, but the world is not yet ready for dynamically invoked services available in public directories. One good sign of the maturing of the SOA market is that SAP, IBM, and Microsoft announced the closure of their public UDDI service towards the end of 2005, citing the 'lack of need' for a public business register. One useful thing that has come out of the joint industry work on UDDI is a realisation that no-one wants to do business entirely automatically, but that UDDI has some useful features for managing a private (but generally open-ended) network of services. The UDDI protocols have been changed substantially in this direction, and have become a useful mature standard for managing information about services (the metadata).

Long before web services protocols got in on the act, SOA was conceived as an evolution of an architectural approach to managing evolutionary change across an enterprise. This is in line with the idea of the IT architect as town planner explored above; SOA provides a methodical approach to identifying, specifying, and using shared services across an enterprise (and outside it in a trading community or similar structure).

SOA can be seen as a response to the failure of many IT software systems to cope with changing business requirements. The service oriented approach to architecture is to consider the business as a set of collaborating components that work together in various, often-changing ways (business processes). Information systems are then planned to support each component and allow flexibility and agility in the way that each component works with others. The information systems themselves are built as a set of business-related services that each provide a well-defined, coherent set of operations. For example a personnel management service might provide a set of operations relating managing staff. The important thing for the organisation as a whole is that all other information systems make use of the common personnel service instead of trying to maintain their own databases of information. Using SOA techniques it is possible to build systems that can be adapted and changed without a major re-procurement or redevelopment exercise.

The ability to provide for change — known variously as 'flexibility' or 'agility' — is generally thought to be a 'Good Thing'. According to the Gartner Group [12], the whole software market is shifting to start providing smaller service-orientated components instead of large monolithic 'do-it-all' systems.

Experience in industry suggests that the organisations that have adopted SOA successfully have also invested in centrally-funded infrastructure to support common services for all systems and applications to use, and in a senior group of technical and business staff who can oversee the process of adoption of SOA and set clear technical direction for diverse application development teams. In other words to do SOA you have to excel at managing your infrastructure (how else are you going to do metering and billing at the service transaction level?) and you have to have a team of IT architects in control of the technical direction of the enterprise.

6 Techniques and Method

This section looks forward to further seminars about the techniques and methods used by IT architects; it is based on the IBM Global Services Method, so while most of this section is generally applicable to all IT architects, there is a bias towards how architects work in an IT services company, running projects for customers.

6.1 Why do architects need a method?

A good method is designed to enable a common language among all practitioners delivering business solutions. This clearly also includes many people who are not architects. The common language of a method enhances the communication between them.

For architects in a services delivery organization like IBM Global Services or Accenture, a method is a fundamental part of shifting the business to repeatable, asset-based services. It provides a mechanism for practitioners to reuse knowledge and assets using a consistent, integrated approach. This is generally better than making it up as you go along.

This common language across all communities and teams to describe planning and delivery of projects is one of the key sources of value from any investment in methods. A method also allows projects to be started more quickly by using project templates and checklists. If the templates also include a set of well-defined work product descriptions, then the method provides a framework for educating the different parts of a large organization to a common standard, and helps people from different parts of the world collaborate.

At the most basic level using a method improves the quality of an architect's work and reduces the risks of omitting some vital step or spending too long in unimportant areas. One of the key attributes of good architects is their willingness to learn from other peoples mistakes and successes. Part of the method will be a set of anti-patterns to avoid, and a set of patterns to reuse, based on good work in the past. In some cases this can even extend to reusing (at least the structure) of whole deliverables.

6.2 What does a method include?

The core of the method will be a set of descriptions of different work products, some blank templates to get you started (and to keep you consistent with an overall style), and some suggestions of how the individual work products might be combined into client deliverables.

A deliverable, in case you ask, is something that you can deliver. It normally takes the form of a document or a presentation produced for and approved by the senior manager at the client who is paying for your work. The formal acceptance of a deliverable is often associated with a payment as part of the overall work. Even when the deliverable is less tangible (like a successful conclusion to system testing) there is generally a document associated with it (for example a test results report) that can physically be signed off.

To give you an idea of what each work product consists of, here is a list selected at random from the IBM Global Services Method, in the Architecture section.

- ARC 115 : Software Distribution Plan
- ARC 117 : Viability Assessment
- ARC 118 : Change Cases
- ARC 119 : Non-Functional Requirements
- ARC 300 : Feasibility Study
- ARC 301 : Current IT Environment
- ARC 302 : Application Function Model
- ARC 303 : Business Roles and Locations
- ARC 304 : Current IT Assessment
- ARC 305 : Data and Function Access and Placement
- ARC 306 : Data Stores
- ARC 307 : Enterprise Information Model
- ARC 308 : Enterprise Technology Framework
- ARC 309 : Principles - Policies and Guidelines
- ARC 310 : Standards

Each one of these will have a supporting document that describes what it is about, why you might add it to your plan, when you can leave it out (and what might happen if you do), some examples from completed work products, and some considerations for estimating how much effort will be involved.

Here for example is part of the description from ARC304:

> The Current IT Assessment work product provides an assessment of the current IT environment. This assessment can take many forms: an assessment of the functional and technical quality of a client's installed applications, databases and/or infrastructure.
>
> With respect to the applications, the assessment is focused on two complimentary views of functional and technical quality (FQ / TQ as it is sometimes called). The former is concerned with both the extent to which the applications support the critical business functions and the quality of that support. The latter is concerned with their technical efficiency, (e.g. their age, stability, accessibility, reliability, maintainability) and the longer term technical potential.
>
> For database assessment, this work product is focused on data quality, (i.e. currency, redundancy, accuracy) and the potential to support the future business strategies.
>
> With respect to the current infrastructure, the assessment is focused on technical efficiency, e.g. stability, reliability, maintainability, age of installed base, support for critical business functions e.g. availability, performance, accessibility, anticipated business growth and the longer term technical potential.
>
> The scope of this type of assessment is usually limited to a subset of the application / database portfolio that is judged to be the most important in terms of their support for the current business operations and their future potential.

Any specific problems and opportunities associated with these applications, databases and infrastructure are captured and documented as input to subsequent activities and work products

Most of the work products will be accompanied by template documents that can be used as a starting point for collecting the results or information gathers as part of doing the work product. Some work products, may also be accompanied by detailed **technique papers**. For the architect these are the crown jewels of a method, the most useful parts.

A technique paper is normally written by one or more very experienced senior architects and represents their combined wisdom distilled from the painful experience of many different projects and client engagements. While most of the method is essentially book keeping (making sure that you don't forget things and that your documents look consistent), the technique papers actually help the architect to do his or her job. They explain the background to a given area, and provide extensive practical help about how to approach and manage a given task. For example, one of the best papers currently in the IBM method is called 'Designing for Availability'. This document contains a number of chapters and appendices, that each provides a different approach to thinking about high availability solutions. These include: one on 'Practice & Process' that provides a high level process to define the scope of the high availability (HA) work; another on 'High Availability Patterns' that provides more detail through a core set of reusable HA patterns across the areas of application, data, network and infrastructure; and others on the of key tools and techniques that can be used to assist in the capture of HA requirements, and the analysis and design of a system for HA, and how to use them with the rest of the method.

As well as specific subject-oriented guidance on using work products together, a method may also include high level patterns that show how to group work products together and how they relate to each other. These capability patterns will include logical maps that shows what input is needed for each work product and where the output that it produces should be used. The patterns also include definitions of the roles that will be needed to deliver the particular project, and a suggested break down of all the work involved. In simple cases, these workload breakdown structures (WBS) can be loaded directly into a project planning tool, although this is rarely a prime responsibility of the IT architect.

In large organizations, a global method will tend to grow over time and become unwieldy and full of duplication. Even in a smaller organization the same drift is often apparent, so it is vital to be professional about managing the method content. This normally means using a proper document management system that publishes each part of the method to a structured web site. A good search engine to index all the content is also invaluable.

Managing all the content on a given project can also be a daunting task, and while many of us manage with large collections of word processor documents and presentations, a formal project support tool is included in some methods to provide a structured database to produce and manage the various written deliverables.

Typical use of all these different parts is generally quite simple, once enough practitioners are familiar with it.

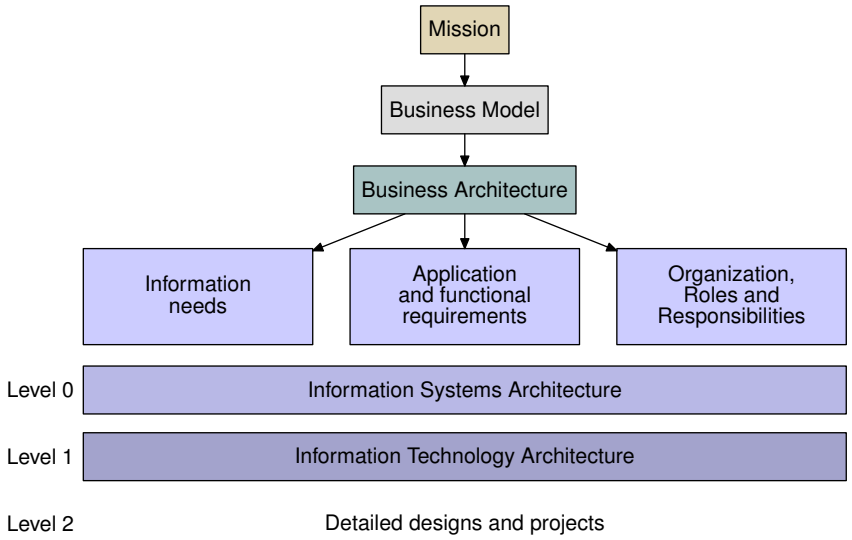
1. Identify the deliverables and associated work products

2. Create or reuse a plan with tasks to build these work products
3. Estimate effort and risk, and refine the plan accordingly
4. Create a statement of work as a summary of the plan
5. Execute the plan, monitoring and modifying as usual

6.3 Architectural Thinking

Architectural Thinking is the name of IBM's education class that introduces architects to using the method to do architecture. It introduces some key artefacts and provides some insight into the core techniques of architectural practice. The artefacts are the 'Architecture Overview Document' and the supporting models that show how the business functions are delivered (the Component Model) and how the actual systems will be run (the Operational Model).

Before introducing these it is helpful to consider the normal context. Architecture is part of a process, part of managing communications and information systems. Every time that we design a new system we need to consider the architecture of the whole enterprise. This is shown in the following diagram.



The upper part of the pyramid (above the line of three boxes) is the domain of the business architects. The mission of a business is what it does, what it wants to do, what it is for. The business model describes how it does these things, how it 'goes to market'. The business architecture defines how it is structured to achieve its goals: the business architecture consists of (at least) the three boxes below it: a set of functional requirements, flanked by a set of information requirements, and a general organizational plan.

The job of the IT architect is to support this business architecture from below (as it were). At the highest level is the Information Systems Architecture, this is normally documented with an Architectural Overview. The highest level of technical architecture document, deliberately kept short (perhaps 20 pages) and comprehensible (good illustrations). It should have the widest possible readership within an organization.

The level below this is the Information Technology Architecture which sets out how the communications and information systems will be used to deliver what the business need. At this level of documentation there are two complementary aspects, which IBM calls the component model and the operational model.

The component model defines the overall structure of the IT systems, how they are broken down into autonomous areas, what common services they will use, the interfaces between them, the design standards they should use, the policies for storing and reading data, the information design, and so on.

The operational model is more concerned with the systems as they will run in production. It includes guidelines for physical designs and types of hardware, for availability and performance, for business resilience, and so on.

Beneath this level we find detailed designs for individual projects: each of these will also have an architectural overview supported by a component model and an operational model, but at this level they focus on how this particular system works, and they refer upwards to the higher level architectures in order to show how they implement the policy and guidance set out there.

How these elements of architectural method are used and combined will be the subject of a much more detailed seminar later in this series. The main aim of them is to provide a structured approach to making decisions about the communications and information systems architecture. Decisions with a project scope are made (and documented) at level 2, while decisions that affect several projects are incorporated in the level 0 documentation and form a key part of the Architecture Overview.

7 Retrospect and prospect

This introductory module has attempted to cover the wide sweep of activities that make up IT architecture. We have reviewed the basic ideas and concepts of the IT architect as the custodian of the ‘big picture’ and the universal communicator, able to explain what is going on in a variety of ways to the different teams that are involved in creating a large IT system. For each team the architect draws up a view that provides enough context for the team to work effectively, but filters out irrelevant details.

We have looked at the different roles that an IT architect can play and the range of talents and knowledge needed to succeed. Good architects combine an ability to understand deep complex technical issues with an enthusiasm and ability to communicate in many different ways. Working in teams is not just something IT architects do, it is something that they fundamentally enable.

We have also reviewed the terms and vocabulary of modern IT architecture, and briefly examined how some of them are applied to contemporary practice. We have concluded our introduction by looking at why architects need a method to guide them through their work and how such methods work.

The final remaining part of this introduction is to look forward to what is coming next; not just in this series of classes but in your future careers.

☞ *Before that let's just examine some misconceptions about architecture, and some positive definitions...* (These are from [13]).

7.1 What architecture is not

- *Architecture is just paper*
- Architecture and design are the same things
- *Architecture and infrastructure are the same things*
- <my favourite technology> is the architecture
- *A good architecture is the work of a single architect*
- Architecture is simply structure
- *Architecture can be represented in a single blueprint*
- System architecture precedes software architecture
- *Architecture cannot be measured or validated*
- Architecture is a science
- Architecture is an art

7.2 What architecture is

- Architecture defines major components
- *Architecture defines component relationships (structures) and interactions*
- Behaviour of components is a part of architecture insofar as it can be discerned from the point of view of another component
- *Every system has an architecture (even a system composed of one component)*
- Architecture defines the rationale behind the components and the structure
- *Architecture definitions do not define what a component is*
- Architecture is not a single structure — no single structure is the architecture

7.3 Career paths

In the IT industry the 'shape' of an IT architect is often discussed. This is nothing to do with the physical condition of aging technical staff. It is to do with breadth and depth of technical knowledge. Deep technical specialists tend to know one area in great detail; such people are very valuable and some of them have very satisfying careers. These people are sometimes called I-shaped. Other people have a superficial knowledge across the whole range of communications and information technology, but no deep specialist knowledge of anything. These people are generally called salesmen. People who make good architects are a mixture of these two types. They have deep specialist skills in at least one area, and they have a broad understanding of the whole field of IT. They are often called T-shaped, although you might be able to think of some better names.

In planning your career you will find it helpful to decide what shape you want to be. You will note that if you want to be an architect it is necessary to develop a deep understanding of one particular area (at least one), if you pick your area at the right stage in your career it can often be very helpful in shaping the opportunities you get and getting your name widely known.

A thorough grounding in the processes of software development is often helpful, as is experience with the physical end of installing and operating machines. A general enthusiasm for technology while not essential is also often a great help.

7.4 Suggestions for further work

Finally here are some suggestions and possibilities for further work to follow up the topics we have explored in this session.

- Take one of the technical discussion topics listed below, and research it. Write up your findings in various forms: a single page of A4 with high level discussion points that succinctly explain the technical area and highlight some architectural issues and decision points; a 10 minute presentation with annotations; a 2000 word essay; a policy statement for the manifesto of a political party (you'll struggle to do this for some of them); a fact sheet for 14–16 year olds; a Wikipedia entry (don't cheat! write one first then compare it to what Wikipedia actually says).
- Working in small groups take one of the technical discussion topics or some other burning technical issue, and present to the seminar group for five minutes about it.
- Re-read the Roy Schulte article [3] and write a 3000 word review and critique of it given the benefit of 10 years of hindsight. Highlight which trends are likely to continue and which predictions have failed to materialize.
- Lead a 30 minute discussion group about the misconceptions and definitions about architecture from [13] stated above.
- Read a different book from the bibliography to this paper every month for a whole year. Record your thoughts and insights in a blog.

The "References" section at the end includes all the books cited in this introduction plus several others that are recommended for useful background.

7.5 Technical discussion topics

These topics are intended for general discussion and for practice in explaining a new technical area to various audiences.

1. What's wrong with the QWERTY keyboard? Describe some developments in computer input devices?
2. What is quantum cryptography?
3. Describe some potential applications for smart cards. What implementation challenges might they face?
4. What is the size of the system management market? Who are the major players?
5. Describe some technologies to help disabled people use the Internet.
6. Give an overview of Unicode.
7. Will future changes to user interface technology change any fundamental architectural disciplines?
8. What is 'push technology'? What is it good for? What are the limits?
9. Describe recent developments in display technology.
10. Compare the advantages and disadvantages of copper versus optical fibre for transmitting telecommunications.
11. What is fibre channel arbitrated loop?
12. Give a short analysis of Cisco's performance in the market since 1995. What do they do and where they are heading?

13. What's the difference between an Application Service Provider (ASP) and an Active Server Page (ASP)?
14. What are the main graphics formats used on the Web?
15. What is a storage area network (SAN)?
16. Where does the term RAID come from? Describe some uses of RAID storage.
17. Describe the advances in Intel's new IA-64 processor family.
18. What are PostScript, PCL and AFP?
19. Describe the business threats posed by three different types of computer 'malware'.
20. Compare the strengths of a traditional UNIX shell as an IDE (integrated development environment) with any GUI development tool set.
21. Can AMD's market share ever overtake Intel's?
22. Describe the difference between Macromedia Flash, Director and Shockwave.
23. What is the market for speech recognition products?
24. How does speech recognition actually work?
25. What is electronic ink, what can you do with it, when do you expect it to be widely available?
26. Set out the strengths and weaknesses of Bluetooth.
27. For how long can you run a PC on batteries? What technologies are used in modern batteries?
28. Other than Unicode, what are the technical architecture issues about Internationalization (I18N) and Localization (L10N)?
29. Describe the major implementation approaches for VPNs.
30. Why is software configuration management important in development?
31. How does satellite data transmission work?
32. Give a short overview of the document management marketplace.
33. Tell us all about WAP. Why do the Americans think it won't work? What's the problem with it?
34. What is SET? How is it related to C/SET?
35. Compare Windows CE, EPOC and PalmOS.
36. Describe the leading back up and recovery systems for UNIX.
37. Explain how ADSL works.
38. Use examples to show how vertical market applications differ from horizontal market applications.
39. What is an enterprise information portal?
40. Give a short overview of the iSeries platform (what used to be called the AS/400).
41. What is ATM networking? What inhibits its rapid market acceptance?
42. Why did Psion withdraw from the PDA market place?
43. What does work flow mean? What are the major work flow products?
44. Explain what makes Internet telephony possible.
45. What is meant by pervasive computing? How might is affect your daily life?
46. Give a technical appreciation of Mac OS X.

References

[1] Richard Scarry. *What do people do all day?* Picture Lions, 1976. ISBN 0007111584.

[2] L. Cherbakov et al. Impact of service orientation at the business level. *IBM Systems Journal*, 44(4):653–668, 2005.

[3] Roy Schulte. Architecture and Planning for Modern Application Styles. Technical Report R-ARCH-104, Gartner Group, 1997. The original paper naming Service Oriented Architecture, and defining the role of the architect as the 'city planner' who determines what common services a 'city' should invest in, and polices new developments to make sure that they use the services appropriately.

[4] Roy Schulte. Clarifying the terms 'Event Driven' and 'Service-Oriented Architecture'. Technical Report G00126127, Gartner Group, 2005. The term 'service-oriented architecture' has evolved in several directions. Developers should understand its multiple meanings and appreciate the distinction between 'service-oriented' and 'event-driven'. These two design models are key to effective distributed architecture.

[5] John Henessey and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996. ISBN 1558603298.

[6] Alistair Cockburn. *Writing Effective Use Cases*. Addison Wesley, October 2000. ISBN 0201702258.

[7] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures*. Addison-Wesley, 2002. ISBN 0201703726. A complete Users' Guide that explains not only the '4+1 views' from the Rational Unified Process, but other approaches as well.

[8] Philippe Kruchten. Book Review — Documenting Software Architectures, August 2002. `http://www-128.ibm.com/developerworks/rational/library/2868.html`.

[9] Fred Brooks. *The Mythical Man-Month*. Addison-Wesley, 1995. ISBN 0201835959.

[10] Martin Fowler. *Analysis Patterns — Reusable Object Models*. Addison Wesley, November 1996. ISBN 0201895420.

[11] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, November 2002. ISBN 0321127420. A handbook for enterprise system developers guiding them through the intricacies and lessons learned in enterprise application development.

[12] Simon Hayward. Positions 2005: Service-Oriented Architecture Adds Flexibility to Business Processes. Technical Report G00126409, Gartner Group, 2005. SOA will transform software from inhibitor to enabler of business change, but requires new investments; subscription software and composite applications will replace packages and monolithic suites.

[13] Philippe Kruchten. *Software Architecture and Iterative Development*. Benjamin Cummings, 2004. ISBN 0805305963.