JOURNAL 1

JOURNAL1 MICROSOFT ARCHITECTS JOURNAL JANUARY 2004 A NEW PUBLICATION FOR SOFTWARE ARCHITECTS

Dear Architect
Adapting to change has always been important to business success and today change is more rapid than ever. Companies merge, markets shift, competitors develop new products, and customers demand change, all with unprecedented speed. In today's business environment, staying ahead means continuously adapting to change – and making change work in your favour. To do that, business leaders need IT systems that support and enable their strategic decisions. There is a growing consensus in the industry that the way to create this kind of adaptive, agile IT architecture is through Web services – discrete units of software, based on industry-standard protocols that interoperate across platforms and programming languages.

Whether or not you build IT systems and Web service-based connectivity using Microsoft® Windows™ and Microsoft® .NET, you still need to connect together a broad range of personal and business technologies. These enable you to access and use important information, whenever and wherever it is needed. The final result you desire is an integrated, cost-effective IT architecture that empowers your business. Information once isolated in back-end systemsis now available to all your employees via streamlined and automated processes that can span multiple systems.

It gives me great pleasure to introduce the very first issue of the new 'Microsoft Architects Journal' – a platform where authoritative software architects from all corners of Microsoft's architect community will discuss the connection between opportunities once out of reach and the solutions that now make them possible.

We hope you will enjoy your journal.

Simon Brown
General Manager,
Developer and Platform
Evangelism Group, Microsoft EMEA

Editorial

By Arvindra Sehmi

Dear Architect
Welcome to the inaugural issue of JOURNAL! Software architecture is a tough thing – a vast, interesting and largely unexplored subject area. As an art, it requires intuition and understanding of well-established architectural disciplines. As an engineering practice, it leads to formation of system models consisting of parts; with descriptions of their shape and form in terms of properties, relationships and constraints. The rationale for their existence often derives from the system requirements. And of course, everyone has or wants to say something about it!

The richness of this topic is one of the reasons we have launched JOURNAL – 'Microsoft Architects Journal'. It will be a platform for thought leadership on a wide range of subjects on enterprise application architecture, design and development. Authors will discuss various business and 'soft' concerns that play a key role in enterprise systems development. It will provide a unique source of information previously not available through any other Microsoft offering.

The responsibilities and required capabilities of the architect vary, depending on the particular role that is being fulfilled during the enterprise solutions development cycle. Typically, during the early business and IT strategy phases, the architect provides a supporting role. This involvement serves to add value in visioning and scoping, helping to reduce complexity and risks, and ensuring the strategy is viable and feasible. The architect also gains knowledge of the business and its aspirations, which will provide the basis for architecture design. The architect can translate between the technology view and business view of the strategy.

As the organisation moves into enterprise and project architecture design phases, the architect role becomes much more significant. The architect is responsible for structuring, modelling and design of the architecture.

Finally, as the organisation moves through subsequent phases to implement, deploy and maintain the solution, the architect performs a guiding and verification role: ensuring quality and architectural implementation conformance to the design. The architect may also optimise the architecture, as the problem domain becomes better understood. In all stages, the architect will perform as part of the assignment team, and frequently a team will fulfil the architecture role. An indicative list of typical architect responsibilities is:

- Support business visioning and scoping activities
- Translate between business and IT requirements
- Communicate with stakeholders, both within business and IT
- Weigh different interests
- Determine solution alternatives
- Create a viable and feasible design
- Choose solutions
- Manage quality
- Manage complexity
- Mitigate risks
- Communicate

This requires a diverse range of skills, including knowledge of architecture design, workgroup and communications skills, and consultancy skills. In this issue of JOURNAL, we reflect the

Keep updated with additional information at http://msdn.microsoft.com/architecture/journal wide variety of architectural issues in today's software industry – from Service-Oriented Architecture and best practices for rule-based systems to the role of Blogging in an enterprise solution. Our authors come from organizations throughout the world, and each offers a unique perspective on software architecture and design.

Andrew Macaulay, a technical architect at Cap Gemini Ernst & Young, writes about enterprise architecture design and the Integrated Architecture Framework, and describes a model for enterprise architecture and its importance in helping software architects understand the business as a whole.

David Sprott and Lawrence Wilkes, analysts at CBDI Forum, provide an insight into Service-Oriented Architecture. In their article they outline some of the principles of architecting solutions with services and emphasize the importance of a service-oriented environment.

Gerke Geurts and Adrie Geelhoed, architects at LogicaCMG, discuss communication patterns and their role in defining business processes and services. They assert that such patterns afford decomposition of business processes into business, informational and infrastructural services and the definition of their dependencies, thus providing a solid basis for enterprise information and application architecture.

Kevin Perera, systems architect with Temenos, will present a pragmatic 'late-bound' approach using metadata descriptions of artifacts at design time, to make development time implementation of his applications extremely flexible.

Dennis Merrit, a principal in Amzi! Inc., discusses the problem of encoding logical rules, and argues for a rules processing engine based approach to business automation in which the rules are abstracted from the process.

Clemens Vasters, an executive team member at newtelligence AG and prominent 'blogger', expounds the merits of Weblogs as a means of sharing knowledge and ideas. He describes the lessons he learned while designing and implementing 'dasBlog' – a Web service based Weblog engine built using Microsoft .NET technologies.

I'm certain you'll find something of interest and value in JOURNAL. We'll be keeping you updated with additional information at [http://msdn.microsoft.com/architecture/journal] where you'll also be able to download the articles for your added convenience.

Finally, if you have an idea for an article you'd like to submit for a future issue of JOURNAL, please send a brief outline and resume to me – asehmi@microsoft.com.

Please also e-mail me any comments on JOURNAL.

Arvindra Sehmi
Architect, Developer and Platform
Evangelism Group, Microsoft EMEA

Enterprise Architecture Design and the Integrated Architecture Framework

By Andrew Macaulay, CGE&Y

Enterprise Architecture in context
Over the past few years, and as software and systems engineering has matured, it has become accepted that there is a clear need for an ‘architectural view’ of systems. This need has grown as a result of the increasing complexity of systems and their interactions within and between businesses. Furthermore, continued pressure to reduce IT costs and deliver real, quantifiable business benefit from solutions necessitate a clear understanding of how systems support and enable the business.

The ‘architectural view’ of systems (both business and IT systems) is defined in the ANSI/IEEE Standard 1471-2000 as: ‘the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution’. Further to this high-level definition, and in the same way as there are different levels of architecture within building (city plans, zoning plans and building plans), it is important to classify business and IT architecture into a number of different levels:

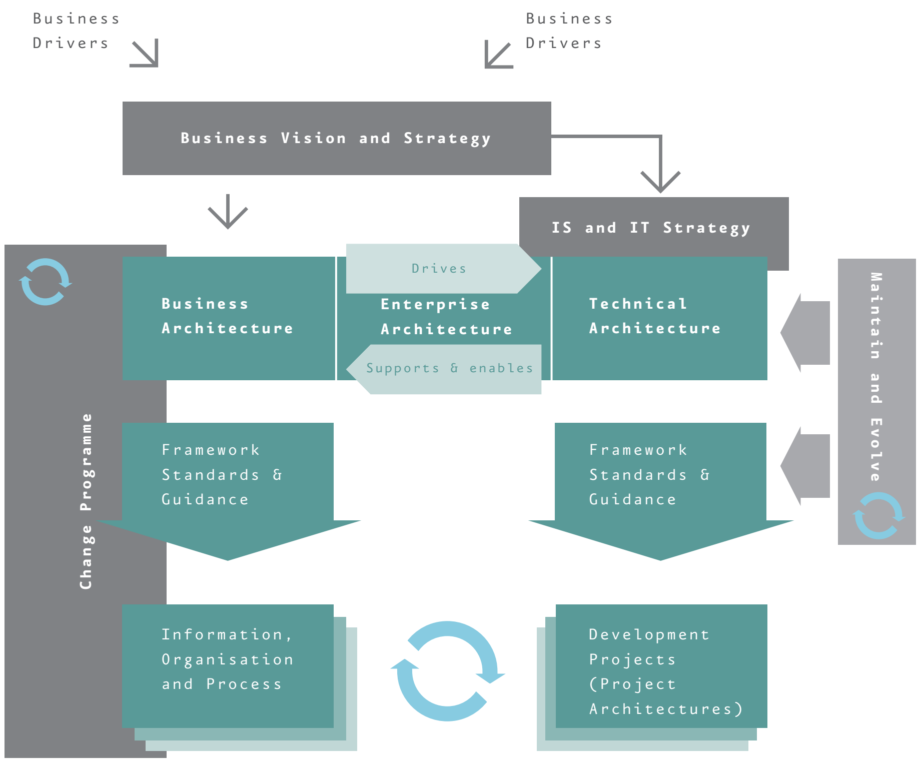
Enterprise Architecture – defining the overall form and function of systems (business and IT) across an enterprise (including partners and organisations forming the extended enterprise), and providing a framework, standards and guidelines for project-level architectures. The vision provided by the Enterprise Architecture allows the development of consistent and appropriate systems across the enterprise with the ability to work together, collaborate or integrate where and when required.

Project-Level Architecture – defines the form and function of the systems (business and IT) in a project or programme, within the context of the enterprise as a whole and not just the individual systems in isolation. This project-level architecture will refine, conform to and work within the defined Enterprise Architecture.

Application Architecture – defines the form and function of the applications that will be developed to deliver the required functionality of the system. Some of this architecture may be defined in the Enterprise and Project-level Architecture (as standards and guidelines) to ensure best-practice and conformance to the overall architecture.

When considering how organisations typically manage business change and IT enablement, traditional approaches to strategic business change use a top down view of the business in terms of its people and processes. However, the traditional software and systems engineering approaches tend to focus on identifying and delivering the specific functionality required to automate a task or activity. Less importance is attached to how the resulting system will interact with other systems and the rest the

Figure 1. Business and Systems Alignment



> “Over the past few years, and as software and systems engineering has matured, it has become accepted that there is a clear need for an ‘architectural view’ of systems.”

business in order to deliver wider business benefit. As a result, there is often a gap between the high level vision and structure of the business, and the systems implemented to support them (in other words the alignment between business and IT is poor).

To bridge this gap, many organisations are developing an enterprise architecture to provide a clear and holistic vision of how systems (both manual and automated) will support and enable their business. An effective enterprise architecture comprises a comprehensive view of the business, including its drivers, vision and strategy; the organisation and services required to deliver this vision and strategy; and the information, systems and technology required for the effective delivery of these services (see Figure 1).

Defining an enterprise architecture is complex, because it encompasses the systems within the context of the whole enterprise. To simplify this, an enterprise architecture is typically structured by considering a business or system as a series of components (or services) with inter-relationships, without having to consider the detailed design within the individual components.

Both the components and their inter-relationships must be viewed in terms of the services that they provide, and the characteristics, such as security, scalability, performance, integration, required of those services. These components can then be grouped by service characteristics, distribution and other business-driven aspects as well as functionality.

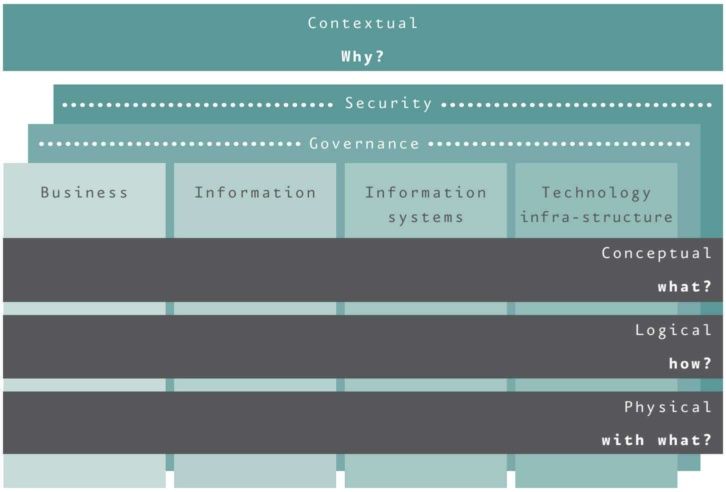
Although enterprise architecture should ideally be designed using a top down approach, many organisations have severe budget constraints for strategic IT initiatives which do not readily offer short-term return on investment or quantifiable business benefit. For this reason, many enterprise architectures are initially created as part of an approved large project or programme. Once in place there are opportunities to refine it further and to start demonstrating benefit and value by providing standards and guidelines for subsequent projects.

Cap Gemini Ernst & Young's Integrated Architecture Framework

Cap Gemini Ernst & Young has, over the past 10 years, developed an approach to the analysis and development of enterprise and project-level architectures know as the Integrated Architecture Framework (IAF). This approach, now its third major revision, has been developed at a global level based on the experience of Cap Gemini Ernst & Young architects on real projects, together with a formal review process including academics. IAF has been successfully used on many hundreds of engagements, both large and small, across the globe.

IAF breaks down the overall problem into a number of the related aspect areas covering *Business* (people and process), *Information* (including knowledge), *Information Systems,* and *Technology Infrastructure,* with two specialist areas addressing the *Governance* and *Security* aspects across all of these. Analysis of each of these areas is structured into four levels of abstraction: *Contextual, Conceptual, Logical* and *Physical,* as shown in *Figure 2*.

Figure 2. Integrated Architecture Framework



Contextual answers the question 'why do we need an architecture?' and 'what is the overall context?'

Conceptual answers the question 'what are the requirements and what is the vision of a solution?'

Logical answers the question 'how are these requirements to be met?'

Physical answers the question 'with what is the solution built?'

This approach allows the pragmatic deployment of the framework in many different scenarios, both by using only the relevant parts of the framework and by supporting iterative working across the streams. This flexibility minimises the traditional effects of a waterfall approach and ensures coherency across the aspect areas. For example, a project architecture using IAF will, in many cases, only need to use sufficient of the Business and Information aspect areas to provide the overall context for the project. An enterprise architecture will concentrate mainly on the contextual, conceptual and logical levels.

The *Contextual* level brings together the business and other drivers, vision and strategy and their resulting priorities into a set of principles all of which are described with their implications and priorities. This comprehensive set of statements is then used in a consistent manner in the decision making process, providing traceability back to the original business drivers, strategy and vision, and demonstrating the required business-systems alignment.

Although much of the work done at this stage is concerned with data gathering, the importance of this stage cannot be overstressed. It will provide the basis for the entire architecture design by creating and documenting an understanding of the scope from an overall business perspective.

The *Conceptual* level details the services and the interactions between these services in support of the principles defined in the Contextual level. As the models defined in the Conceptual level are service-based (that is they do not detail specific products or standards), they remain stable unless the business itself fundamentally changes its vision and objectives; providing a solid foundation from which the logical architecture can be derived. Key decisions taken at this level include:

- What areas of the business to use IT to support?
- Which overall business architecture (e.g. moving to a front-office, mid-office, back-office model) will be used?
- How systems will reflect the organisation/business architecture, the level to which department systems are consolidated into a suite of core applications or are allowed departmental flexibility with a central integration service?

The *Logical* level describes the solution as product-independent services or components, includes a clear definition of the integration and collaboration contracts between these services or components. By remaining independent of products, this level of the architecture can remain relatively stable. It can change to reflect top-down changes, including new fundamental business models (for example a move towards a Customer-Centric model), as well as bottom-up changes, such as opportunities for technology enablement (such as CRM) or fundamental changes in technology paradigm (for example Services Architecture or Grid). Through this, the impact of change at the business or technology level can be assessed in a clear and consistent manner.

Key decisions taken at this level include:

- How should the logical components be grouped (for example, providing a multi-channel customer logon service with separate channel-specific/optimised authentication components alongside a common authentication support component using a central directory)?
- How will logical components be shared across systems (these components could be presented as Web Services)?
- How a central integration hub supports the various business systems, or the way in which collaboration tools are used alongside databases applications and integration to support virtual team?

Typically, at the logical level, there might be more than one way to approach the solution (which reflects the various drivers, for example cost, flexibility, security, manageability). The key decision at this level is then to select (with the business) the solution alternative that delivers the services required, in a way that best addresses the guiding principles.

The *Physical* level details the design principles, standards and guidelines, including component grouping in critical areas as well as deployment models. This provides the framework within which the detailed design can be undertaken, as well as selection criteria (not functional specifications) for products to be either developed or purchased.

It is at this level that solution frameworks and architectures such as the Microsoft Systems Architecture (MSA) can be used at this level to accelerate development of the physical architecture, improve the quality of the architecture (by using proven solutions) and reduce project risks.

Examples of key decisions taken at this level are:

– Which physical components which will be part of a package solution (e.g. using physical components from the ERP solution).
– What additional components will be required around this package, and the standards and guidelines for developing these components (e.g. language, tools, etc.)?
– What are the standards and detailed product selection criteria for the infrastructure products to be deployed? – leading onto a candidate list for selection. If there is a clear product or vendor strategy, this candidate list becomes the product standards to be taken forward.

Communication and Architecture Governance

Because of the large number of potential stakeholders, the enterprise architecture needs to be communicated at many different levels, using the appropriate visual representations and language: For senior management, the architecture must show how the business goals and drivers will be supported, and how benefit can be derived. The focus of business users is more on their own individual business areas and is more functionally-biased. IT management and staff will want to focus on the technical components, including how they will be able to provide the required levels of support. Project-level architects will be concerned with the standards and guidelines which will provide re-use opportunities or impose constraints on their individual designs.

Programmes and projects must conform to the enterprise architecture to ensure that business benefits can be realised, and that the systems and software engineering activities can benefit from the analysis already done in defining the enterprise architecture. Furthermore, as with all architectures, the enterprise architecture requires ongoing maintenance, especially around the more physical areas such as technical standards. This ensures that the enterprise architecture remains valid and relevant to the business as it changes. The enterprise architecture should be under the control of an enterprise-wide governance function that ensures its maintenance and verifies the ongoing conformance of systems. Even where, for clear and justified business reasons, conformance is not possible, this function is then able to make sure that the business understands the real costs of implementing non-conformant systems, for example increased running costs or lack of future flexibility.

Linking Architecture and Design

As the use of the term 'architecture' has grown within business and IT, there have been many areas of confusion: for example, in many organisations, architecture and design are seen as being the same thing. It is Cap Gemini Ernst & Young's view that this is not

Table 1. Comparing Architecture and Design

	Delivers	Does Not Deliver
Architecture	– Non-functional requirements – Functional scope and responsibilities (who does what) – Key design and product choices – High level design – Design constraints	– Prototypes – Comprehensive functional requirements – Detailed data analysis – Built and implemented systems
Design	– Functional requirements and how they will be met – Detailed data analysis and data model as necessary – System design documentation – Built and implemented systems	– Solution Vision – Comprehensive and traceable non-functional requirements – Security and governance architectures

> "Programmes and projects must conform to the enterprise architecture to ensure that business benefits can be realised."

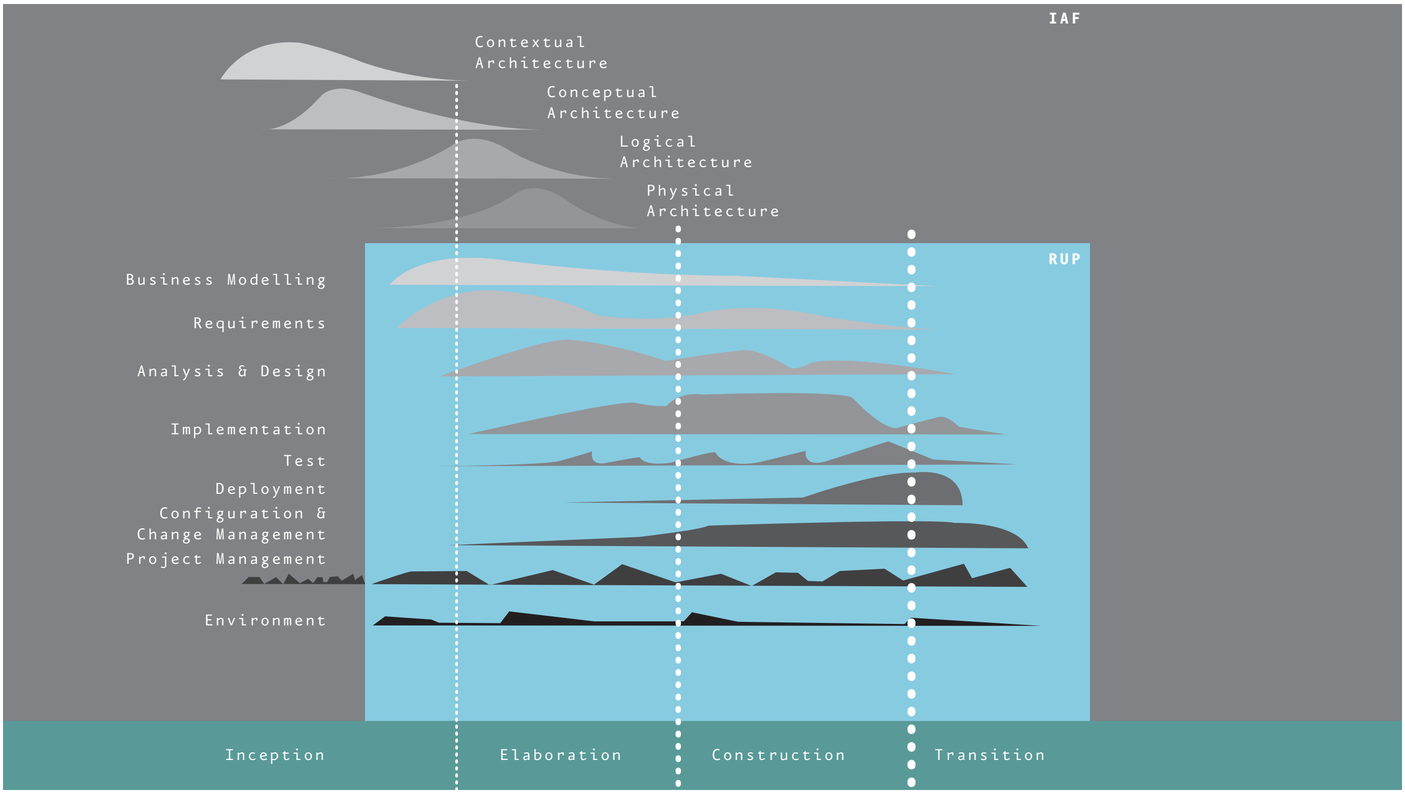
the case because design and architecture offer different and complimentary perspectives on the solution – in fact, Cap Gemini Ernst & Young use IAF and the Rational Unified Process (RUP; a software development approach) on projects to deliver a complete design approach from architecture to code. Table 1 shows the comparison between architecture (IAF) and design (RUP, including application architecture):

As with the architecture of buildings, software architecture and (detailed) design are, in fact, part of an overall 'design continuum' required to deliver a complete solution. Aligning IAF and RUP processes provides a comprehensive and consistent framework in support of this. Furthermore, the enterprise architecture will provide much of the context and other inputs required by the project architecture, whilst the project architecture will cater for the unique requirements of the solution. In projects with significant potential risk, especially on complex or large projects, the use of the architectural approach at project-level will mitigate many of the risks by ensuring that there is a clear and holistic understanding of the overall context of the solution including external systems and drivers that may affect the solution.

With IAF, a project-level technical architecture uses the same basic approach as the enterprise architecture, albeit with different levels of detail, with more focus on the logical and physical level of information, information systems, technology infrastructure, governance and security aspects (using the context and business architecture defined in the enterprise architecture).

As a result, the output from the enterprise architecture can be used directly as the input into the project-level technical architecture. From the project-specific technical architecture, the detailed and specific design principles, guidelines, standards, and

Figure 3. IAF and RUP



constraints, which then guide the detailed software and systems engineering design activities, can be derived. In the case of IAF, the output from the project-level technical architecture can be mapped onto RUP design artefacts such as business use cases, system use cases, and non-functional specification. These mappings typically are not one-to-one, but do provide traceability through from the architecture to the physical detailed design, as well as helping accelerate the overall design process from architecture through to delivered systems. This helps accelerate the design/development effort whilst continuing to mitigate project risks.

Summary

Enterprise architectures are becoming more important today as the level of complexity and inter-operation between systems and business increases, and as there is even more need for business-system alignment and cost-effective use of IT to deliver business benefit. Enterprise architecture (and project-level technical architecture) provides valuable input into application architecture and detailed design by helping architects understand the business as a whole and by placing the solution being designed into the overall business and technical context within which the project is being delivered.

The key objectives of an enterprise architecture are to *understand …*
- The relevant parts of whole business, in context (incl. external partners)
- The end-to-end processes (including external processes/actors)
- Non-functional requirements (including security & governance)

which *results in a solution that …*
- Supports the non-functional requirements.This may need specific component organization to support, for example, specific cross-domain security requirements, or a service-based approach to provide the required flexibility.
- Is seen in the context of the whole business and end-to-end processes.For example, service level objectives may exist for overall transaction times that span more than one business – understanding the limitations of this allow these measures to be refined.
- Links to, and is traceable to, the business principles so that the impact of changes, some of which may result from the design stage, can be evaluated in business terms.
- Drives, contextualises, and constrains the design. The design for the application or infrastructure will need to be governed by the architecture in order to fully deliver the complete solution including the non-functional requirements.
- Is clearly scoped, understood and clearly defines the responsibilities of each element.

and *provides the rationale …*
- For decisions, standards and product selections that support the business goals and drivers.

Further Reading

Cap Gemini Ernst & Young Technology Services
http://www.cgey.com/technology

Services Architecture
http://www.cgey.com/technology/sa/index.shtml

Adaptive IT
http://www.cgey.com/adaptive/solutions/adaptive-it-overview.shtml

Andrew Macaulay
andrew.macaulay@cgey.com

Andrew Macaulay joined Cap Gemini Ernst & Young as a Technical Architect in 1993 following ten years as a Technical Consultant. He has been an Architect and Technical Consultant on many major engagements, both at an Enterprise and the Project level, and has been instrumental in developing, delivering and training Cap Gemini Ernst & Young's approach to architecture, Integrated Architecture Framework.

Understanding Service Oriented Architecture

By David Sprott and Lawrence Wilkes, CBDI Forum

If there was a hit parade of IT acronyms, Service Oriented Architecture, or SOA, would surely be number one. Yet for all the media comment, how many really understand what SOA is? How does it affect what architects, CIOs, project managers, business analysts and lead developers do? In this article we provide a concise explanation that we anticipate will baseline the subject.

Introduction

It seems probable that eventually most software capabilities will be delivered and consumed as services. Of course they may be implemented as tightly coupled systems, but the point of usage – to the portal, to the device, to another endpoint, and so on, will use a service based interface. We have seen the comment that architects and designers need to be cautious to avoid everything becoming a service. We think this is incorrect and muddled thinking.

It might be valid right now given the maturity of Web Service protocols and technology to question whether everything is implemented using Web services, but that doesn't detract from the need to design everything from a service perspective. The service is the major construct for publishing and should be used at the point of each significant interface. Service Oriented Architecture allows us to manage the usage (delivery, acquisition, consumption, and so on) in terms of, and in sets of, related services. This will have big implications for how we manage the software life cycle – right from specification of requirements as services, design of services, acquisition and outsourcing as services, asset management of services, and so on.

Over time, the level of abstraction at which functionality is specified, published and or consumed has gradually become higher and higher. We have progressed from modules, to objects, to components, and now to services. However in many respects the naming of SOA is unfortunate. Whilst SOA is of course about architecture, it is impossible to constrain the discussion to architecture, because matters such as business design and the delivery process are also important considerations. A more useful nomenclature might be Service Orientation (or SO). There are actually a number of parallels with object orientation (or OO) and component based development (CBD):

- Like objects and components, services represent natural building blocks that allow us to organize capabilities in ways that are familiar to us.
- Similarly to objects and components, a service is a fundamental building block that
  a. Combines information and behaviour.
  b. Hides the internal workings from outside intrusion.
  c. Presents a relatively simple interface to the rest of the organism.
- Where objects use abstract data types and data abstraction, services can provide a similar level of adaptability through aspect or context orientation.
- Where objects and components can be organized in class or service hierarchies with inherited behaviour, services can be published and consumed singly or as hierarchies and or collaborations.

For many organizations, the logical starting place for investigating Service Oriented Architecture is the consideration of Web services. However Web services are not inherently service oriented. A Web service merely exposes a capability which conforms to Web services protocols. In this article we will identify the characteristics of a well formed service, and provide guidance for architects and designers on how to deliver service oriented applications.

Principles and Definitions

Looking around we see the term or acronym SOA becoming widely used, but there's not a lot of precision in the way that it's used. The World Wide Web Consortium (W3C) for example refers to SOA as *'A set of components which can be invoked, and whose interface descriptions can be published and discovered'*. We see similar definitions being used elsewhere; it's a very technical perspective in which architecture is considered a technical implementation. This is odd, because the term architecture is more generally used to describe a style or set of practices – for example the style in which something is designed and constructed, for example Georgian buildings, Art Nouveau decoration or a garden by Sir Edwin Lutyens and Gertrude Jekyll .

CBDI believes a wider definition of Service Oriented Architecture is required. In order to reach this definition, let's start with some existing definitions, and compare some W3C offerings with CBDI recommendations. We'll begin by looking at definitions of basic Service concepts.

Service

- A Component capable of performing a task. A WSDL service: A collection of end points (W3C).

– A type of capability described using WSDL (CBDI).

A Service Definition
– A vehicle by which a consumer's need or want is satisfied according to a negotiated contract (implied or explicit) which includes Service Agreement, Function Offered and so on (CBDI).
A Service Fulfillment
– An instance of a capability execution (CBDI).

Web service
– A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a format that machines can process (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with XML serialization in conjunction with other Web-related standards (W3C).
– A programmatic interface to a capability that is in conformance with WSnn protocols (CBDI).
From these definitions, it will be clear that the W3C have adopted a somewhat narrower approach to defining services and other related artefacts than CBDI. CBDI differs slightly insofar as not all Services are Components, nor do they all perform a task. Also CBDI recommends it is useful to manage the type, definition and fulfilment as separate items. However it is in the definition of SOA that CBDI really parts company with the W3C.

Service Oriented Architecture:
– A set of components which can be invoked, and whose interface descriptions can be published and discovered (W3C).

CBDI rejects this definition on two counts: First the components (or implementations) will often not be a set. Second the W3C definition of architecture only considers the implemented and deployed components, rather than the science, art or practice of building the architecture. CBDI recommends SOA is more usefully defined as:

The policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards based form of interface. (CBDI)

CBDI defines SOA as a style resulting from the use of particular policies, practices and frameworks that deliver services that conform to certain norms. Examples include certain granularity, independence from the implementation, and standards compliance. What these definitions highlight is that any form of service can be exposed with a Web services interface. However higher order qualities such as reusability and independence from implementation, will only be achieved by employing some science in a design and building process that is explicitly directed at incremental objectives beyond the basic interoperability enabled by use of Web services.

SOA Basics
It's would be easy to conclude that the move to Service Orientation really commenced with Web services – about three years ago. However, Web services were merely a step along a much longer road. The notion of a service is an integral part of component thinking, and it is clear that distributed architectures were early attempts to implement Service Oriented Architecture. What's important to recognize is that Web services are part of the wider picture that is SOA. The Web service is the programmatic interface to a capability that is in conformance with WSnn protocols. So Web services provide us with certain architectural characteristics and benefits – specifically platform independence, loose coupling, self description, and discovery – and they can enable a formal separation between the provider and consumer because of the formality of the interface.

Service is the important concept. Web Services are the set of protocols by which Services can be published, discovered and used in a technology neutral, standard form.

In fact Web services are not a mandatory component of a SOA, although increasingly they will become so. SOA is potentially much wider in its scope than simply defining service implementation, addressing the quality of the service from the perspective of the provider and the consumer. You can draw a parallel with CBD and component technologies. COM and UML component packaging address

> "Higher order qualities such as reusability and independence from implementation, will only be achieved by employing some science in a design and building process."

components from the technology perspective, but CBD, or indeed Component Based Software Engineering (CBSE), is the discipline by which you ensure you are building components that are aligned with the business. In the same way, Web services are purely the implementation. SOA is the approach, not just the service equivalent of a UML component packaging diagram.

Many of these SOA characteristics were illustrated in a recent CBDI report[1], which compared Web services published by two dotcom companies as alternatives to their normal browser-based access, enabling users to incorporate the functionality offered into their own applications. In one case it was immediately obvious that the Web services were meaningful business services – for example enabling the Service Consumer to retrieve prices, generate lists, or add an item to the shopping cart.
In contrast the other organization's services are quite different. It implemented a general purpose API, which simply provides Create, Read, Update, and Delete (CRUD) access to their database through Web services. While there is nothing at all wrong with this implementation, it requires that users understand the underlying model and comply with the business rules to ensure that your data integrity is protected. The WSDL tells you nothing about the business or the entities. This is an example of Web services without SOA.

SOA is not just an architecture of services seen from a technology perspective, but the policies, practices, and frameworks by which we ensure the right services are provided and consumed.

So what we need is a framework for understanding what constitutes a good service. If, as we have seen in the previous example, we have varying levels of usefulness, we need some Principles of Service Orientation that allow us to set policies, benchmarks and so on.

We can discern two obvious sets here:
- Interface related principles – Technology neutrality, standardization and consumability.
- Design principles – These are more about achieving quality services, meeting real business needs, and making services easy to use, inherently adaptable, and easy to manage.

Interestingly the second set might have been addressed to some extent by organizations that have established mature component architectures. However it's certainly our experience that most organizations have found this level of discipline hard to justify. While high quality components have been created perhaps for certain core applications where there is a clear case for widespread sharing and reuse, more generally it has been hard to incur what has been perceived as an investment cost with a short term return on investment.

However when the same principles are applied to services, there is now much greater awareness of the requirements, and frankly business and IT management have undergone a steep learning curve to better understand the cost and benefits of IT systems that are not designed for purpose. Here we have to be clear – not all services need all of these characteristics; however it is important that if a service is to be used by multiple consumers, (as is typically the case when a SOA is required), the specification needs to be generalized, the service needs to be abstracted from the implementation (as in the earlier dotcom case study), and developers of consumer applications shouldn't need to know about the underlying model and rules. The specification of obligations that client applications must meet needs to be formally defined and precise and the service must be offered at a relevant level of granularity that combines appropriate flexibility with ease of assembly into the business process.

Table 1 shows principles of good service design that are enabled by characteristics of either Web services or SOA.

If the principles summarized in Table 1 are complied with, we get some interesting benefits:
- **There is real synchronization between the business and IT implementation perspective**. For many years, business people haven't really understood the IT architecture. With well designed services we can radically improve communications with the business, and indeed move beyond alignment and seriously consider convergence of business and IT processes.
- **A well formed service provides us with a unit of management that relates to business usage.** Enforced separation of the service

[1] Service Based Packaged Applications http://www.cbdiforum.com/secure/interact/2003-07/service_based_pkd_apps.php3

provision provides us with basis for understanding the life cycle costs of a service and how it is used in the business.

- **When the service is abstracted from the implementation it is possible to consider various alternative options for delivery and collaboration models.** No one expects that, at any stage in the foreseeable future, core enterprise applications will be acquired purely by assembling services from multiple sources. However it is entirely realistic to assume that certain services will be acquired from external sources because it is more appropriate to acquire them. For example authentication services, a good example of third party commodity services that can deliver a superior service because of specialization, and the benefits of using a trusted external agency to improve authentication.

Process Matters

As indicated earlier, CBDI advises that good SOA is all about style – policy, practice and frameworks. This makes process matters an essential consideration.

Whilst some of the benefits of services might have been achieved by some organizations using components, there are relatively few organizations that rigorously enforce the separation of provision and consumption throughout the process. This gets easier with services because of the formality of the interface protocols, but we need to recognize that this separation needs managing. For example it's all too easy to separate the build processes of the service and the consumer, but if the consumer is being developed by the same team as the service then it's all too easy to test the services in a manner that reflects understanding of the underlying implementation.

With SOA it is critical to implement processes that ensure that there are at least two different and separate processes – for provider and consumer.

However, current user requirements for seamless end-to-end business processes, a key driver for using Web Services, mean that there will often be clear separation between the providing and consumer organizations, and potentially many to many relationships where each participant has different objectives but

Table 1. Web services and SOA

Enabled by Web services	***Technology neutral***	Endpoint platform independence.
	Standardized	Standards based protocols.
	Consumable	Enabling automated discovery and usage.
Enabled by SOA	***Reusable***	Use of Service, not reuse by copying of code/implementation.
	Abstracted	Service is abstracted from the implementation.
	Published	Precise, published specification functionality of service interface, not implementation.
	Formal	Formal contract between endpoints places obligations on provider and consumer.
	Relevant	Functionality presented at a granularity recognized by the user as a meaningful service.

nevertheless all need to use the same service. Our recommendation is that development organizations behave like this, even when both the providing and consuming processes are in-house, to ensure they are properly designing services that accommodate future needs

For the consumer, the process must be organized such that only the service interface matters, and there must be no dependence upon knowledge of the service implementation. If this can be achieved, considerable benefits of flexibility accrue because the service designers cannot make any assumptions about consumer behaviours. They have to provide formal specifications and contracts within the bounds of which consumers can use the service in whatever way they see fit. Consumer developers only need to know where the service is, what it does, how they can use it. The interface is really the only thing of consequence to the consumer as this defines how the service can be interacted with.

Similarly, whilst the provider has a very different set of concerns, it needs to develop and deliver a service that can be used by the Service Consumer in a completely separate process. The focus of attention for the provider is therefore again the interface – the description and the contract. Another way of looking at this is to think about the nature of the collaboration between provider and consumer. At first sight you may think that there is a clear divide between implementation and provisioning, owned by the provider, and consumption, owned by the consumer. However if we look at these top level processes from the perspective of collaborations, then we see a very different picture.

What we have is a significant number of process areas where (depending on the nature of the service) there is deep collaboration between provider and consumer. Potentially we have a major reengineering of the software delivery process. Although we have two primary *parties* to the service based process, we conclude there are three major *process areas* which we need to manage. Of course these decompose, but it seems to us that the following are the primary top level processes.

- **The process of delivering the service implementation.**
  - 'Traditional' Development
  - Programming
  - Web Services automated by tools

- **The provisioning of the service – the life cycle of the service as a reusable artefact**.
  - Commercial Orientation
  - Internal and External View
  - Service Level Management

- **The consumption process.**
  - Business Process Driven
  - Service Consumer could be internal or external
  - Solution assembly from Services, not code
  - Increasingly graphical, declarative development approach
  - Could be undertaken by business analyst or knowledge worker

The advantage of taking this view is that the collaborative aspects of the process are primarily contained in the provisioning process area. And the provisioning area is incredibly important because the nature of the agreement has a major influence on the process requirements. There are perhaps two major patterns for designing consumer/provider collaborations:

- **Negotiated – Consumer and Provider jointly agree service**
  When new services are developed though, there is an opportunity for both provider and consumer to agree what and how the services should work. In industries where there are many participants all dealing with each other, and where services are common to many providers, it is essential that the industry considers standardizing those services. Examples include:
  - Early adopters
  - New Services
  - Close partners
  - Industry initiative – forming standards
  - Internal use

- **Instantiated – This is it. Take it or leave it**
  One party in the collaborative scenario might simply dictate the services that must be used. Sometimes the service will already exist. You just choose to use it, or not. Examples include:
  - Dominant partner
  - Provider led – Use this service or we can't do business
  - Consumer led – Provide this service or we can't do business
  - Industry initiative – standards compliance
  - Existing system/interface

Architectures

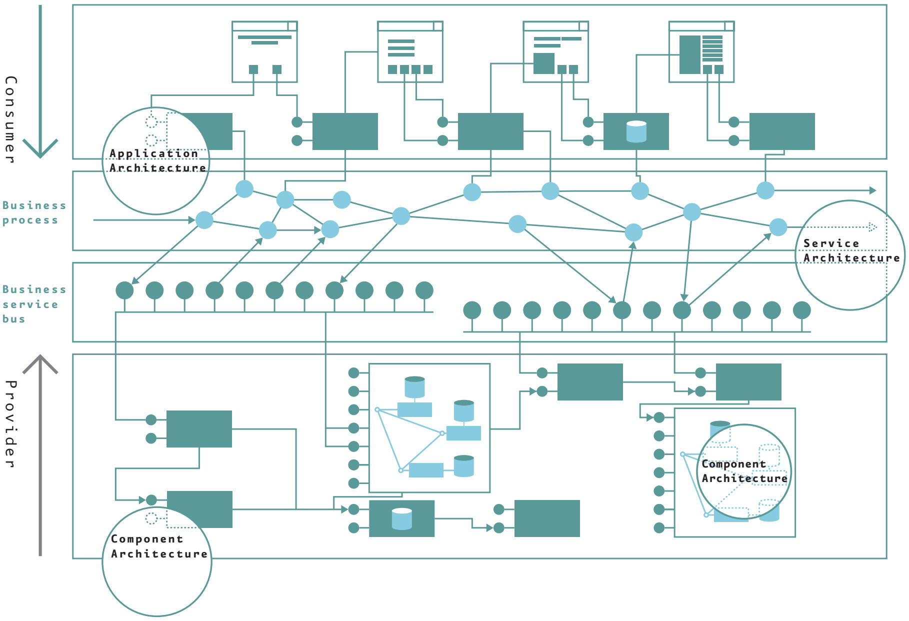
This process view that we have examined at is a prerequisite to thinking about the type of architecture required and the horizons of interest, responsibility and integrity. For SOA there are three important architectural perspectives as shown in *Figure 1*.

– **The Application Architecture.** This is the business facing solution which consumes services from one or more providers and integrates them into the business processes.

– **The Service Architecture.** This provides a bridge between the implementations and the consuming applications, creating a logical view of sets of services which are available for use, invoked by a common interface and management architecture.

– **The Component Architecture.** This describes the various environments supporting the implemented applications, the business objects and their implementations.

Figure 1. Three Architectural Perspectives



These architectures can be viewed from either the consumer or provider perspective. Key to the architecture is that the consumer of a service should not be interested in the implementation detail of the service – just the service provided. The implementation architecture could vary from provider to provider yet still deliver the same service. Similarly the provider should not be interested in the application that the service is consumed in. New unforeseen applications will reuse the same set of services.

The consumer is focused on their application architecture, the services used, but not the detail of the component architecture. They are interested at some level of detail in the general business objects that are of mutual interest, for example provider and consumer need to share a view of what an order is. But the consumer does not need to know how the order component and database are implemented.

Similarly, the provider is focused on the component architecture, the service architecture, but not on the application architecture Again, they both need to understand certain information about the basic applications, for example to be able to set any sequencing rules and pre and post conditions. But the provider is not interested in every detail of the consuming application.

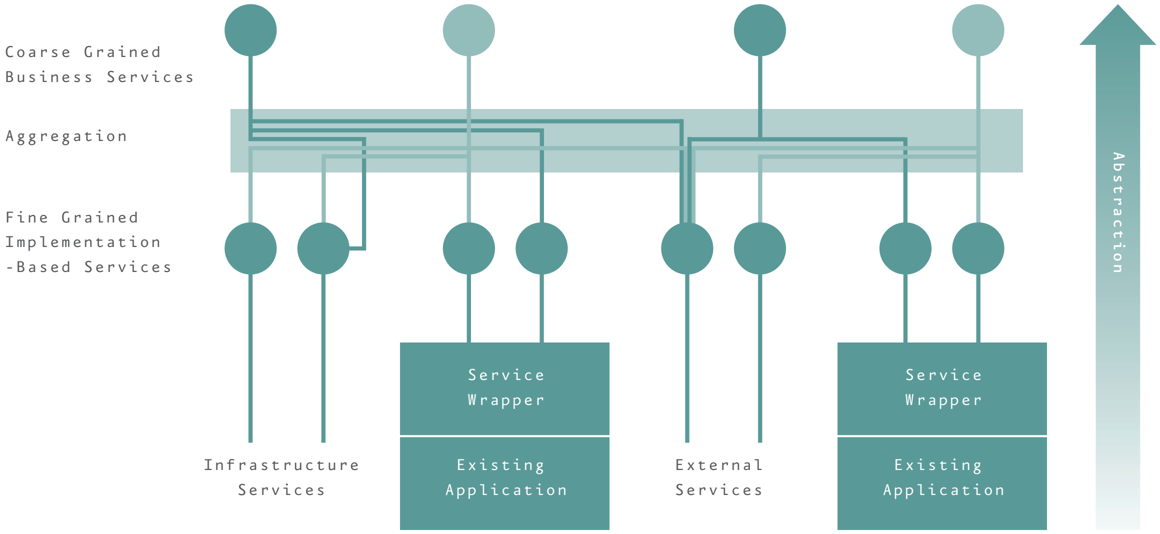
The Service Architecture

At the core of the SOA is the need to be able to manage services as first order deliverables. It is the service that we have constantly emphasized that is the key to communication between the provider and consumer. So we need a Service Architecture that ensures that services don't get reduced to the status of interfaces, rather they have an identity of their own, and can be managed individually and in sets.

CBDI developed the concept of the Business Service Bus (BSB) precisely to meet this need. The BSB is a logical view of the available and used services for a particular business domain, such as Human Resources or Logistics. It helps us answer questions such as:

– What service do I need?
– What services are available to me?
– What services will operate together? (common semantics, business rules)
– What substitute services are available?

Figure 2. Levels of Abstraction



– What are the dependencies between services and versions of services?

Rather than leaving developers to discover individual services and put them into context, the Business Service Bus is instead their starting point that guides them to a coherent set that has been assembled for their domain.

The purpose of the BSB is so that common specifications, policies, etc can be made at the bus level, rather than for each individual service. For example, services on a bus should all follow the same semantic standards, adhere to the same security policy, and all point to the same global model of the domain. It also facilitates the implementation of a number of common, lower-level business infrastructure services that can be aggregated into other higher level business services on the same bus (for example, they could all use the same product code validation service). Each business domain develops a vocabulary and a business model of both process and object.

A key question for the Service Architecture is 'What is the scope of the service that is published to the Business Service Bus?' A simplistic answer is 'At a business level of abstraction'. However this answer is open to interpretation – better to have some heuristics that ensure that the service is the lowest common denominator that meets the criteria of business, and is consumer oriented, agreed, and meaningful to the business. The key point here is that there is a process of aggregation and collaboration that should probably happen separately from the implementing component as illustrated in *Figure 2*. By making it separate, there is a level of flexibility that allows the exposed service(s) to be adjusted without modifying the underlying components. In principle, the level of abstraction will be developed such that services are at a level that is relevant and appropriate to the consumer. The level might be one or all of the following:

- Business Services
- Service Consumer Oriented
- Agreed by both Provider and Consumer
- Combine low level implementation based services into something meaningful to business
- Coarser Grained
- Suitable for External Use
- Conforms to pre-existing connection design

The SOA Platform

The key to separation is to define a virtual platform that is equally

David Sprott
david.sprott@cbdiforum.com
David Sprott, CEO and Principal Analyst, CBDI Forum is a software industry veteran, a well-known commentator and analyst specializing in advanced application delivery. Since 1997 he has founded and led CBDI, an independent analyst firm and think-tank, providing a focus for the industry on best practice in business software creation, reuse and management.

relevant to a number of real platforms. The objective of the virtual platform is to enable the separation of services from the implementation to be as complete as possible and allow components built on various implementation platforms to offer services which have no implementation dependency.

The virtual SOA platform comprises a blueprint which covers the development and implementation platforms. The blueprint provides guidance on the development and implementation of applications to ensure that the published services conform to the same set of structural principles that are relevant to the management and consumer view of the services.

When a number of different applications can all share the same structure, and where the relationships between the parts of the structure are the same, then we have what might be called a common architectural style. The style may be implemented in various ways; it might be a common technical environment, a set of policies, frameworks or practices. Example platform components of a virtual platform include:

- Host environment
- Consumer environment
- Middleware
- Integration and assembly environment
- Development environment
- Asset management
- Publishing & Discovery
- Service level management
- Security infrastructure
- Monitoring & measurement
- Diagnostics & failure
- Consumer/Subscriber management
- Web service protocols
- Identity management
- Certification
- Deployment & Versioning

The Enterprise SOA

The optimum implementation architecture for SOA is a component-based architecture. Many will be familiar with the concepts of process and entity component, and will understand the inherent stability and flexibility of this component architecture, which provide a one to one mapping between business entities and component implementations. Enterprise SOA (ESOA) brings the two main threads – Web services and CBD (or CBSE) – together. The result is an enterprise SOA that applies to both Web services made available externally and also to core business component services built or specified for internal use. It is beyond the scope of this article to explore ESOA in more depth. For more on this topic there is a five part CBDI Report Series on Enterprise SOA[2].

Summary

The goal for a SOA is a world wide mesh of collaborating services, which are published and available for invocation on the Service Bus. Adopting SOA is essential to deliver the business agility and IT flexibility promised by Web Services. These benefits are delivered not by just viewing service architecture from a technology perspective and the adoption of Web Service protocols, but require the creation of a Service Oriented Environment that is based on the following key principals we have articulated in this article;

- Service is the important concept. Web Services are the set of protocols by which Services can be published, discovered and used in a technology neutral, standard form.
- SOA is not just an architecture of services seen from a technology perspective, but the policies, practices, and frameworks by which we ensure the *right* services are provided and consumed.
- With SOA it is critical to implement processes that ensure that there are at least two different and separate processes – for provider and consumer.
- Rather than leaving developers to discover individual services and put them into context, the Business Service Bus is instead their starting point that guides them to a coherent set that has been assembled for their domain.

For further guidance on planning and managing the transition to Web Services and SOA, CBDI are providing the 'Web Services Roadmap', a set of resources that are freely available at http://roadmap.cbdiforum.com/

Lawrence Wilkes
lawrence.wilkes@cbdiforum.com
Lawrence Wilkes, Technical Director and Principal Analyst, CBDI Forum is a frequent speaker, lecturer and writer on Web Services, Service Oriented Architecture, Component Based Development and Application Integration approaches. Lawrence has over 25 years experience in IT working both for end user organizations in various industries, as well as for software vendors and as a consultant.

[2] http://www.cbdiforum.com/secure/interact/2003-03/foundation.php3

Business Process Decomposition and Service Identification using Communication Patterns

By Gerke Geurts and Adrie Geelhoed, LogicaCMG

Introduction

Organisations use information and communication technology (ICT) as a means to reach their objectives. Paradoxically, many organisations find that yesterday's business ICT solutions hinder them in reaching today's objectives. An organisation might be able to modify, wrap or replace existing ICT solutions to meet today's requirements, but how can it avoid today's solutions once again becoming tomorrow's problems?

Many business ICT solutions tend to lack flexibility to deal with changing business requirements and technology. Business requirements change as organisations adjust their strategies, business processes and internal structures to deal with changes in their environment (for example because of competition or legislation) or because they choose to merge or outsource activities. New technologies are introduced to obtain and maintain competitive advantage or because older technologies become too cumbersome and expensive to support.

A main reason for the lack of sufficient flexibility in business ICT solutions is the failure to consider long-term dynamics of business and technology during solution development. Should enterprise fortune-telling therefore become a valued discipline within development projects? Not necessarily. To build ICT solutions that provide long-lasting support for organisations, it is essential to have models that help us to understand the structure and dynamics of the organisations we are trying to support.

Many of the today's organisation and business modelling approaches are based on 'best practices' but to a certain extent remain a black art. What is often missing is a stable theoretical foundation such as the DEMO (Dynamic Essential Modelling of Organisations) framework [Dietz 2002]. By looking at organisations from a communication perspective, DEMO recognises reoccurring communication patterns between people and proceeds to describe how these patterns combine to form business processes.

DEMO provides business analysts and ICT solution architects with tools to decompose the business processes into elementary business transactions. Each business transaction identifies a business role that operates as a miniature supply chain providing a well-defined service to other business roles. Thus when providing automated support to business processes, it makes sense to decompose business process logic according to the same patterns that occur in a purely human world. Therefore, the DEMO approach can be used to identify process-oriented business services in service-oriented and component-based ICT solution architectures.

In this article we will first discuss on how organisations are made up of cooperating people who use communication to align their actions. By studying how people use communication to align their actions, we will encounter the business transaction communication pattern. We will then describe how business transactions are composed to form business processes and enable the identification of business roles. Each business role provides a well-defined service for other business roles within or outside of the organisation. By studying how business roles exchange and remember organisation we will encounter organisational roles that provide informational and infrastructural services. This will result in a technology-independent information architecture that consists of business, informational and infrastructural services and describes their dependencies. We will conclude this article with some remarks regarding the realisation of services using human resources and/or technology.

Organisations

An *organisation* is a group of people who cooperate to achieve common objectives. In all but the simplest organisations no single person is able to perform all the work that must be done to achieve the common goals, so members specialise themselves and must cooperate for the organisation to be effective. To improve cooperation, organisations tend to formalise the roles that members play within the organisation and to agree upon common procedures to coordinate the work performed by different roles.

A *business process* is an ordered execution of activities by people playing organisational roles in order to produce goods or provide services that has add value in the organisation's environment or to the organisation itself. Each activity on its own can be regarded as a decomposable process, and as a sub-process of the containing process [Eriksson and Penker 2000:71]. For example, a sales process may contain a delivery activity that

> "Many business ICT solutions tend to lack flexibility to deal with changing business requirements and technology."

contains nested activities for the selection of a transport provider and the transport itself.

A person may play one or more *roles* within an organisation. Each role represents the elementary authority and responsibility a person must have in order to perform a particular production act. In practice, roles do not usually correspond directly with organisational functions. Often organisational functions map to multiple roles and roles may be played by various organisational functions. For example, 'secretary' and 'claim handler' may be organisational functions within a social service. Both the secretary and the claim handler are allowed to refuse incomplete benefit requests, but only the claim handler is allowed to grant benefits to claimants. So both organisational functions can play the 'claim admission' role, but only one organisational function can play the 'claim granting' role.

The roles participating in a business process perform two types of acts: production acts and coordination acts [Dietz 2002]. *Production acts* contribute to the realisation of goods and/or services for the environment of an organisation. The result of a production act can be material or immaterial. The manufacturing, storage and transport of goods are examples of material production acts. A verdict by a judge or a decision to grant a benefit claim are examples of immaterial products.

Different roles must cooperate when they are dependent on goods or services that are produced by other roles within or outside the organisation. By performing *coordination acts* people enter into and honour commitments towards each other regarding the execution of certain production acts [Dietz 2002]. An example of a commitment is an account manager promising to extend a credit limit when requested to do so by a client.

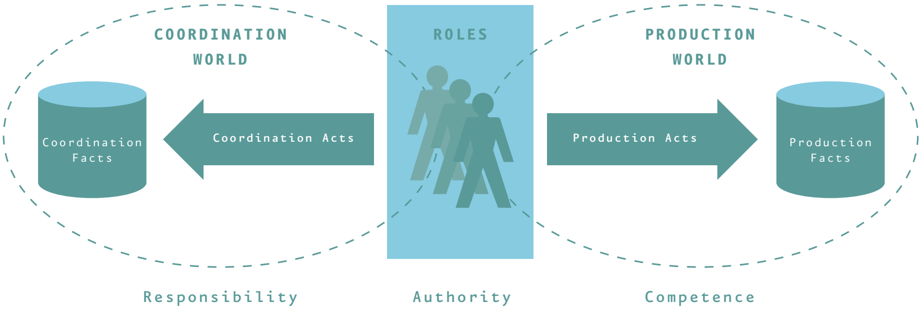
A successful production act causes a change in the 'production world' (the creation of a good or service), which is recorded as a *production fact*. Similarly, a successful coordination act causes a change in the 'coordination world' (the creation or compliance with a commitment), which is recorded as a *coordination fact*. Examples of production facts are 'John's club membership has started to exist' and 'product P has been transported to location L'. Example of coordination facts are 'John has requested Ferdinand to start John's club membership' or 'supplier representative S has promised customer representative C to transport product P to location L'. This principle is shown in *Figure 1*.

Communication

A coordination act is performed by one role (the performer) and directed to another role (the addressee). Because people are not able to read one another's minds, the performer must communicate with the addressee to share his thoughts.

To introduce some communication (and information) concepts we will use an example. A bank client wants his account manager to arrange a higher credit limit on his bank account. To communicate with his account manager, the client formulates his desire for a higher credit limit (mental state of speaker) in a message using a language that both he and the bank manager understand (common language). He then converts the message into perceivable signs by speaking out the message. Sound waves in the air (a communication channel) surrounding the client and the account manager transport the signs to the account manager. The account manager hears the signs and reconstructs the message. She subsequently interprets the message to determine its meaning (mental state of the hearer). She understands that her client wants to increase his credit limit, but as he has a private and a joint account she asks him which account is involved. The client indicates he wants to adjust the credit limit on his private

Figure 1. Coordination and Production Worlds



account. Once she understands completely what her client wants her to do, she checks his credit rating and then promises him she will perform the requested action. Both parties understand that she has committed herself to increasing the credit limit on his private account and that she is expected to comply with her commitment (common social culture).

The example illustrates that communication takes place by the exchanging of *messages*. However, the example also shows that communication is more than the mere exchange of information. When communicating, people try to influence each other's behaviour. By everything they say, they do something [Reijswoud and Dietz 1999:5], i.e. they perform *communicative acts*. This principle is the main tenet of the Language/Action Perspective (LAP) [Austin 1962; Habermas 1981; Searle 1969], a methodology that considers communication from an action perspective.

Messages or communicative acts have a form (syntax) and a meaning (content in the form, i.e. information). The form of a message consists of the language the message is expressed in and the substance (e.g. sound waves, electric charges, light pulses) that carries the message [Dietz and Mallens 2001]. The meaning of a message consists of an intention, a fact and a time-for-completion [Reijswoud and Dietz 1999:13]. A fact specifies a certain action or state of affairs (for example, the credit limit of client account 1-345 is €2000). The intention of a message reflects how a message is intended to be taken by the receiver (e.g. request, promise). Depending on their intention, messages can be grouped in five families:

– *Assertives* commit the speaker to something being the case (for example stating);
– *Directives* try to get the hearer to do something expressed in the message (for example asking, requesting and commanding);
– *Commissives* commit the speaker to one or more actions in the future (for example promising);
– *Declaratives* bring about a new state of affairs by merely declaring it (for example declaring);
– *Expressives* express the attitudes and/or feelings of the speaker about a state of affairs (for example apologising).

The time-for-completion of a message specifies how long the combination of the fact and the intention are valid. In day-to-day conversation the time-for-completion tends to be implicit and a default value of 'now' or 'as soon as possible' is assumed. *Figure 2* shows the form and content of a sample message.

A communication process is successful when mutual understanding is reached, i.e. when the mental state of the speaker and the generated mental state in the hearer correspond [Reijswoud and Dietz 1999:14]. To achieve mutual understanding the speaker and the hearer may ask each other for clarification. Communication is only complete when the hearer confirms he has understood the speaker.

A *conversation* is a sequence of communicative acts between two people with a particular goal [Dietz and Mallens 2001]. Technically, conversations consist of communicative acts that share the same fact and time-for-completion (see *Figure 3*) [Reijswoud and Dietz 1999:20]. Within business processes two kinds of conversations are relevant. *Performative conversations* aim at letting other people do something. *Informative conversations* aim at the sharing of existing knowledge. The main difference between performative and informative conversations, is that performative conversations are about the creation of new (production) facts, whereas informative conversations are about the distribution of knowledge. Examples of informative and performative conversations are shown in *Figure 3*.

Business Events

A coordination act is a communicative act directed from the performer of the coordination act to an addressee. As a communication act consists of an intention, a fact and a time-for-

Figure 2. Message form and content

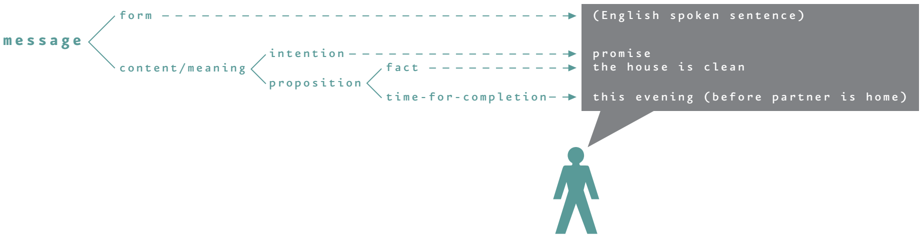


Figure 3. Informative and performative conversation examples

Informative conversation (natural language):	**Performative conversation (natural language):**
Harry: what is the capital of the Netherlands? **Sally**: Amsterdam	**Person W**: The house will be clean when I come back home tonight, won't it? **Sally H**: Off course it will be.
Informative conversation (communicative acts):	**Performative conversation (communicative acts):**
Harry: asks: Sally: What is the capital of the Netherlands? **Sally: States: Harry:** Amsterdam	**Person W: requests: Person H:** The house is clean tonight

completion, coordination acts can be fully specified as follows [Dietz 2002]:

```
<performer>: <intention>:
<addressee>: <fact>: <time-
for-completion>
```

A coordination act is always about a production act, or more precisely about the production fact, whose creation marks successful completion of the production act. The fact in the specification of a coordination act therefore is a production fact.

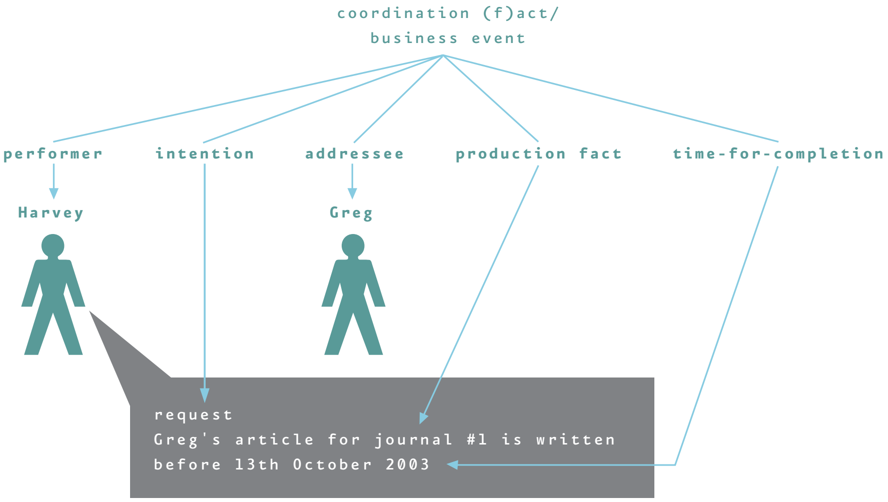
Examples of concrete coordination acts are:

```
Mark: request: Esther:
credit limit of account
1-345 is €2000:tomorrow

Greg: promise: Harvey:
Greg's article for journal
#1 is written:13/10/2003
```

The successful completion of a coordination act is recorded as a coordination fact, which can be specified in the same way as the coordination act. For the performer of a coordination act, the coordination fact represents his/her expectation that the addressee will perform certain actions. From the perspective of the addressee the coordination fact is a 'business event', an event within the organisation or its environment that triggers the organisation to perform some action [Eriksson and Penker 2000:74]. An example of coordination (f)acts and business events is shown in *Figure 4*.

Figure 4. Structure of coordination (f)acts or business events



The Business Transaction Pattern

When studying conversations between people who are trying to coordinate their actions, the communicative acts that make up the conversation appear to follow the same conversation pattern, regardless of the business context. Dietz and Mallens [2001] describe this pattern as follows:

First of all, one agrees upon what is to be achieved. Next the execution takes place in the production world, for example, the shipment of goods ordered. The commitment still stands, however, until the result is accepted between the parties involved, meaning that there is a third phase. It is this pattern that we call a 'business transaction', the elementary block of a business process.

The three phases of the business transaction pattern are shown in *Figure 5*.

A business transaction involves two roles; a customer or *initiator* role starts the business transaction, and a supplier or *executor* role actually

Figure 5. Activity diagram for business transaction pattern

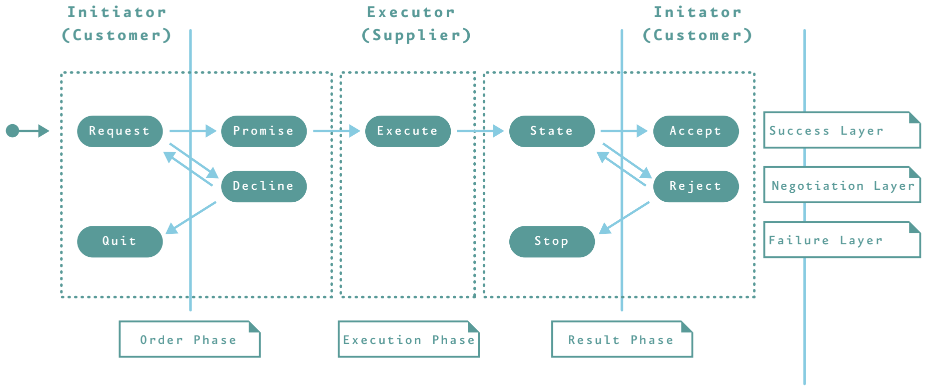
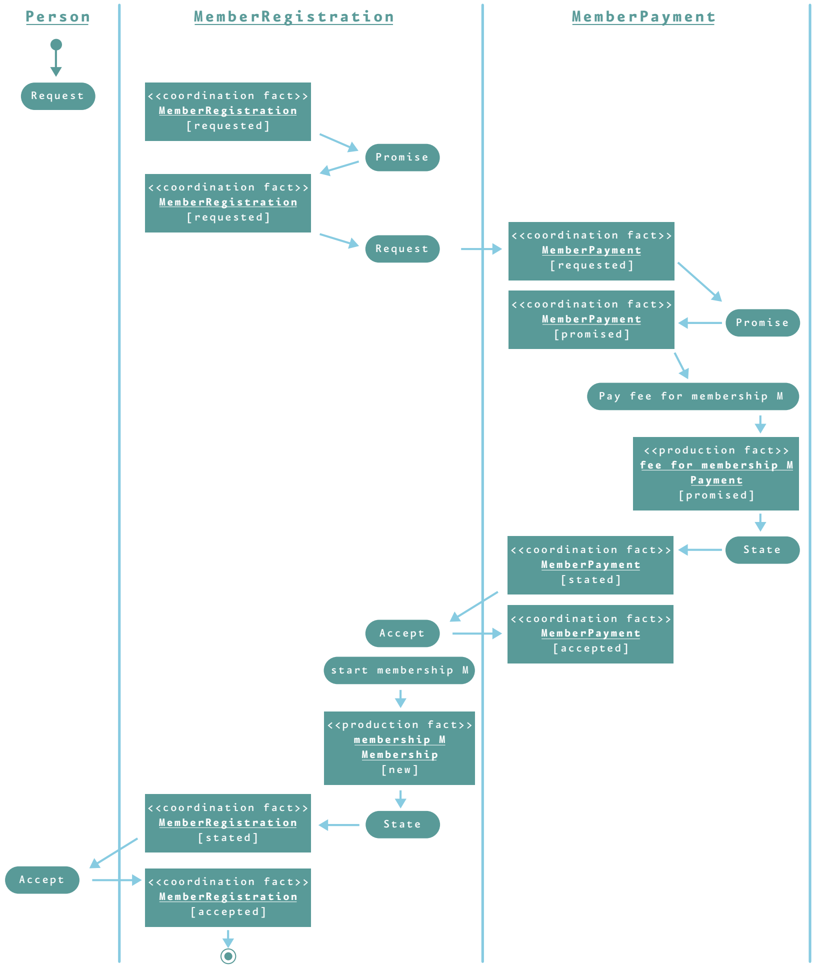


Figure 6. Example business process [Dietz 2002]



performs the intended action. The first phase of a business transaction, the order phase, is a performative conversation that starts with a request from the customer to perform a particular action and ends with a promise by the supplier. During the execution phase the supplier performs the requested action. The transaction is concluded with a performative conversation in the result phase, which starts with a statement from the supplier that the requested action has been performed and finishes when the customer accepts that the action actually has been performed as originally agreed.

The amount of interaction (negotiation) that takes place between customers and suppliers in the order and result phases can vary strongly. In practice there are instances where certain steps are taken implicitly and no communication occurs at all. For example, the expiry of the period in which a customer can return bought products to a shop can be regarded as the implicit acceptance by the customer that the shop has fulfilled its obligations. In other situations long negotiations may take place before suppliers promise certain actions or customers accept the outcome of a production action. The ICT industry (unfortunately) provides plenty of examples in this respect. Reijswoud and Dietz [1999:98-108] describe the business transaction pattern in greater detail.

Business Processes and Roles

In each of the three phases in a business transaction, new business transactions can be initiated whose outcomes are required to continue with the original transaction. In this way

> “The amount of interaction (negotiation) that takes place between customers and suppliers in the order and result phases can vary strongly.”

it is possible to create arbitrary complex structures of nested and chained business transactions. Dietz [2002] defines a business process as a structure of causally interconnected transactions for delivering a particular final product to the environment. In other words, a business process starts with a top-level business transaction which is initiated by a role in the environment of the organisation and which may include other business transactions. These child transactions produce intermediate goods or services (production facts) that are required to successfully complete the parent transaction.

Figure 6 shows an example business process for the registration of library members. The business process consists of two business transactions. The 'member registration' business transaction is initiated by a person who wants to become a library member and executed by employees of the library who are authorised to register new members, i.e. they are allowed to play the 'member registration' role. A person must pay before being registered as member. For this reason the 'member registration' role initiates a second business transaction to obtain payment of the membership fee. The executor of the 'membership payment' transaction is the 'membership payment' role.

Though the 'membership payment' role usually is played by the same person who initiates the membership registration transaction, this is not necessarily the case, e.g. an altruistic friend or relative might pay the fee. This particular example illustrates an important principle: every business transaction has by definition one corresponding *business role*, which represents the elementary amount of authority a person must have to be an executor of the transaction. Dietz [2002] explores the subject of authorisation in greater detail.

Each business role that is responsible for the execution of a particular business transaction operates as an elementary supply chain. Actors in a particular business role can act as supplier in one business transaction and as a consumer in an arbitrary number of subordinate business transactions (see *Figure 7*). Every business role provides an elementary and well-defined business service and may consume one or more other elementary business services offered by other business roles.

Up until this point in our analysis of how organisations achieve their objectives, we have focused on how organisation members 'talk' to each other to align their acts. In the following sections we will concentrate on what people do between coordination actions.

Work Items and Action Rules

An organisation springs to life when a business event occurs – that is, when a coordination fact is created. The coordination fact is created within a particular business transaction instance and signals there is work to be done by one of the roles that participate in the transaction, unless the coordination fact signals the end of the business transaction instance (for example a business transaction has reached the 'accepted' or 'stopped' state).

Every business role has a *work item list* that contains prioritised entries for all coordination facts that the role must respond to. Whether a coordination fact results in a *work item* for the initiator or the executor of a business transaction depends on the intention of a coordination fact. For example, a request results in a work item for the executor, whereas a promise creates a work item for the initiator.

An *action rule* (shown in *Figure 8*) specifies the actions that must be performed by a certain business role in response to a particular coordination fact [Dietz 2002; Reijswoud and Dietz 1999:117-128]. Though action rules are procedures for business roles, these

Figure 7. Business roles

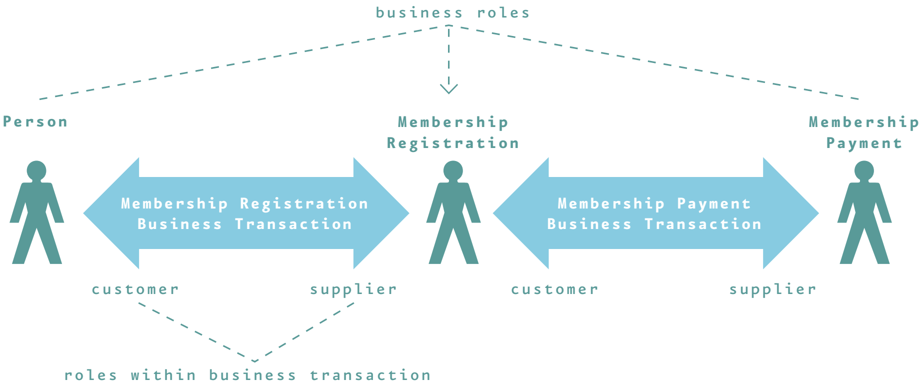

business roles
Person
Membership Registration
Membership Payment
Membership Registration Business Transaction
Membership Payment Business Transaction
customer
supplier
customer
supplier
roles within business transaction

Figure 8. Examples of action rules [Dietz 2002]

Business role	Work item	Action Rule
Membership Registration	**on** requested MembershipRegistration(M)	**with** person P is member of new membership M **if** Age(P) > minimal age for membership **and** number of members < maximum number **then** promise MembershipRegistration(M) **else** decline MembershipRegistration(M) **end if** **end with**
Membership Registration	**on** promised MembershipRegistration(M)	request MembershipPayment(M) **with** fee(M) is remaining_fee(M)
Membership Registration	**on** accepted MembershipRegistration(M)	**if** promised MembershipRegistration(M) **then** <decide to start membership M> state membership M has started **end if**

roles always remain responsible for taking well-considered and socially acceptable decisions, even if that means deviating from the procedures! This is a fundamental reason why it is not possible to fully automate the execution of action rules. When automating action rules, the people who are responsible for the execution of these rules must be able to intervene in and overrule the automated procedures.

The daily life of a person in a business role consists of selecting the work item with the highest priority from his/her work item list and performing the applicable action rule. The execution of an action rule always ends with the performance of one or more coordination acts to pass the buck on to the next role that must play its part in the business process. The coordination act may be preceded by one or more actions to retrieve or calculate facts and/or a production act. provides some examples of action rules written in pseudo code.

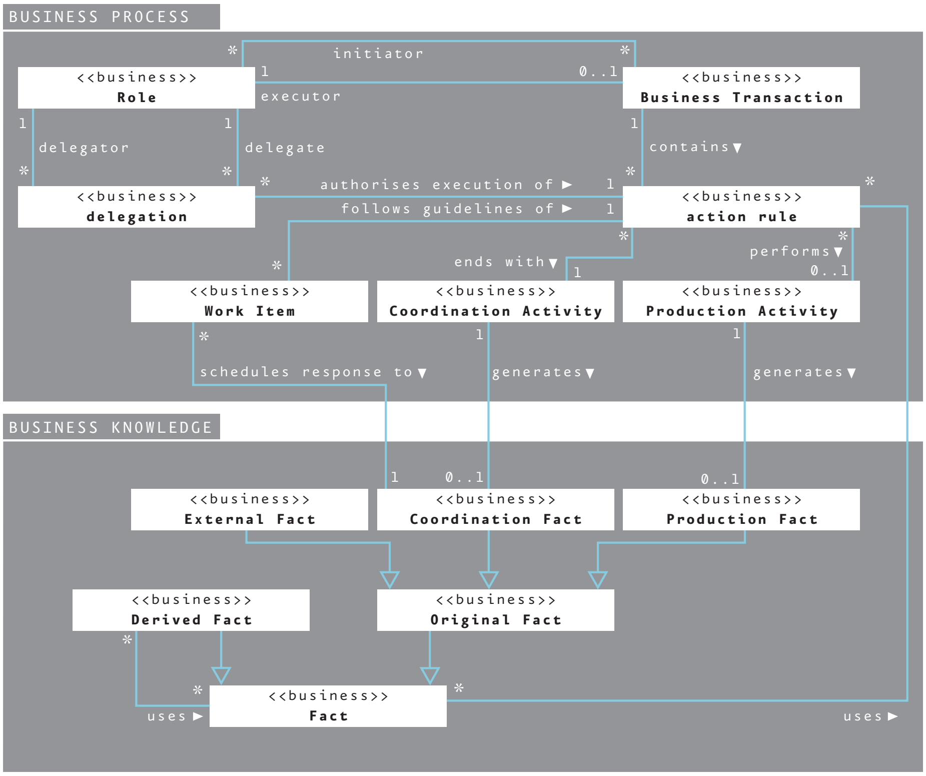
Business Knowledge

The participating roles in a business transaction must have knowledge about applicable business rules (restrictions), production facts and coordination facts in order to be able to act responsibly. The restriction that someone can only rent a video if he has paid all previously completed rentals is an example where knowledge of production facts is needed. Video store employees need knowledge of coordination facts (in this case unfinished video return transactions) to be able to comply with the restriction that a customer may not rent any videos if he/she still has any unreturned videos.

In many cases knowledge of external rules and facts is also required. Legislation and standards are examples of external rules on how acts are to be performed. In any stage of a business transaction external knowledge may be required. For example, during the assessment of a benefit request, a social service may require additional information about the income, housing situation, household and health of an applicant. As the actual facts that must be known about an applicant might depend strongly on his or her situation, the applicant only has to provide a limited amount of

> “The participating roles in a business transaction must have knowledge about applicable business rules (restrictions), production facts and coordination facts in order to be able to act responsibly.”

Figure 9. Business process and business knowledge concepts



information in the benefit request. Additional information will be requested from the applicant and/or other organisations when necessary. *Figure 9* shows business process and business knowledge concepts.

Three Layers of Services

We can view the actions that take place within a business process at three levels of abstraction: the business, informational and infrastructural level. So far we have focused on the business level. At this level we see business roles that are played by *social actors;* actors who are aware of the social implications of their actions. These actors are responsible for the coordination of production acts by entering into and complying with mutual commitments. They are also responsible for the execution of production acts.

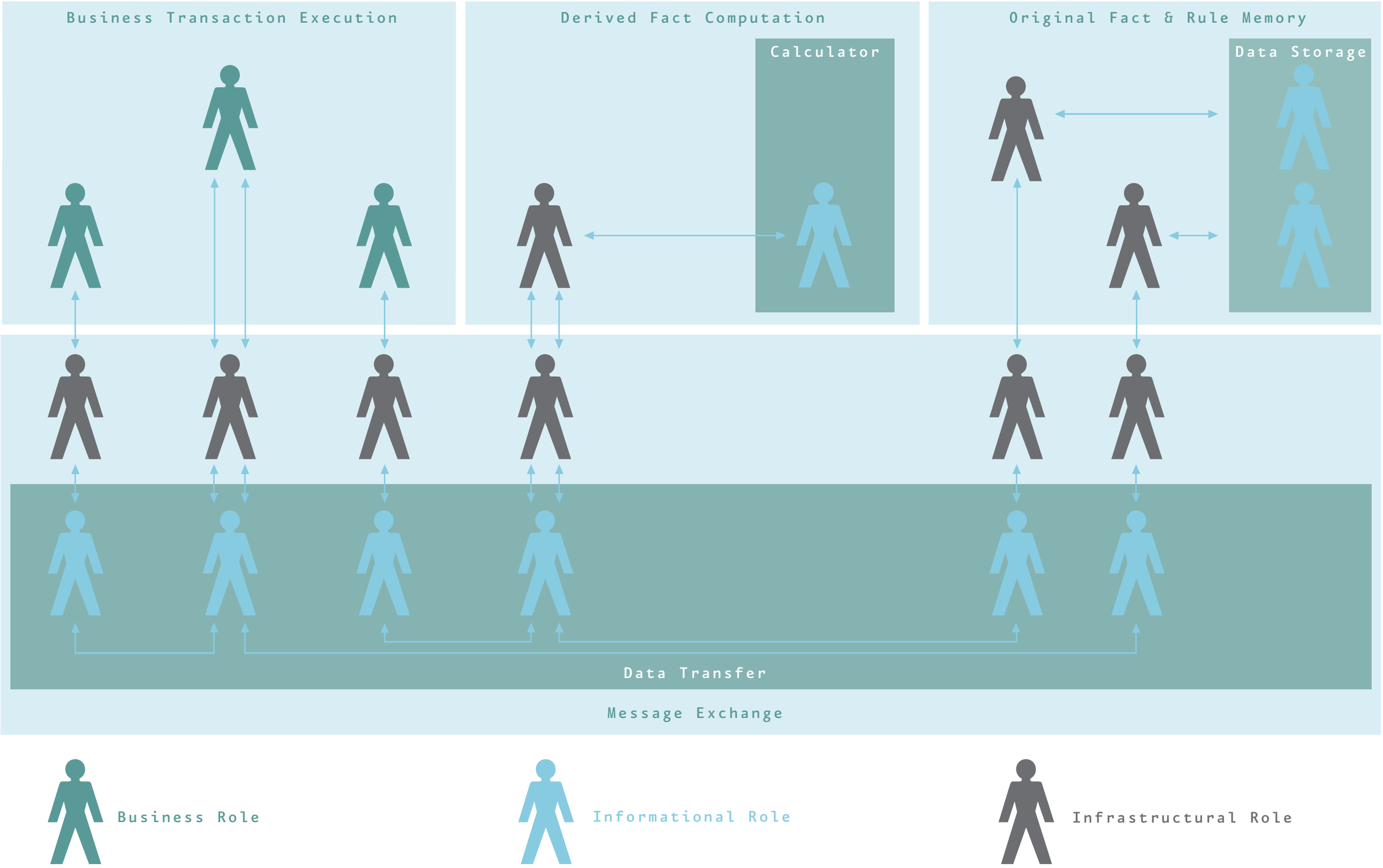
Social actors use the services of *rational actors* to memorise and remember knowledge and to exchange messages with other actors. Rational actors can perform logical operations on knowledge, but have no awareness of the social context they operate in. They play roles at the *informational level*, as they are responsible for the gathering, remembering, providing and computing of knowledge. We can see what knowledge is remembered and distributed by these actors, but cannot see how the storage and transmission takes place. That is, we can see the meaning of documents and messages but cannot see their form.

Rational actors depend on the services of *formal / physical actors* to produce, distribute, store, copy and destroy data or documents that contain knowledge. These actors play *infrastructural roles*: they deal with data and documents but do not show any interest in their meaning.

The actors at the business, informational and infrastructural level provide three types of services, layered on top of each other. At the business level we find business transaction execution services that coordinate other business transaction execution services and informational services to produce a particular good or service. At the informational level we distinguish three categories of informational services: communication services enable the exchange of messages between business roles, computation services offer facilities to compute derived facts and information management services provide long-term memory of (production, coordination and external) facts and business rules. Informational services depend on infrastructural services for the storage, distribution and calculation of data. The dependencies between these roles are shown in *Figure 10*.

The realisation of services at all three levels of abstraction can vary from being fully manual to highly automated. Human beings are able to perform roles at all levels, whereas machines are very capable to perform infrastructural and informational tasks,

Figure 10. Business role support by informational and infrastructural roles



but have absolutely no social awareness. It is for this reason that machines cannot carry the responsibility for performing actions in a socially acceptable manner. In the end there must always be a person who is responsible for the execution of business transactions.

A human actor who plays a certain business role can delegate parts of the execution of business transactions to a machine, but he/she will always remain responsible for the result and have to stand by to deal with exceptional situations. For example, an e-commerce web site may handle the majority of orders without any human intervention. However, a customer who does not receive his order will contact a sales person to resolve the problem.

Conclusion

By analysing how and why communication takes place between cooperating persons, we have encountered the business transaction communication pattern. We have seen how instances of this pattern are chained together to form business processes. The understanding that business processes are composed of business transactions enables us to perform the reverse action; the decomposition of business processes into business transactions.

Each business transaction identifies one business role, which represents the elementary amount of authority and responsibility to execute the transaction. These business roles are the building blocks for authorisation policies within organisations.

Each business role can be viewed as an elementary supply chain and provides a well-defined service. By studying how these services are realised we have encountered the need for additional informational and infrastructural services to enable storage and exchange of information.

The decomposition of business processes into business, informational and infrastructural services and the definition of their dependencies provide a solid basis for enterprise information and application architecture. As technology has not played a role in the decomposition of business processes into services, the resulting services are technology-independent. The degree to which services are automated may vary strongly, but this has no influence on the information architecture.

When implementing a service, we would recommend hiding both the chosen implementation technology from other interacting services as well as the degree of automation within the service. This approach provides an organisation with the flexibility to change the technology itself as well as the way it is applied in business process implementations.

We started this article with the question how to avoid today's solutions becoming tomorrow's headaches. Focusing on communication patterns that stay the same regardless of technology and business process changes certainly seems like a good place to start.

Further Reading

More information on DEMO can be found on the DEMO web site (http://www.demo.nl). The DEMO handbook [Reijswoud and Dietz 1999] provides detailed information on the theoretical background of DEMO and also describes a business modelling approach.

References

Austin 1962
J L Austin, How to do things with words, Harvard University Press, Cambridge MA, 1962

Dietz 2002
J L G Dietz, The Atoms, Molecules and Fibers of Organizations, Data & Knowledge Engineering, 2003, internet: www.demo.nl/documents/2003-DKE.pdf

Dietz and Mallens 2001
J L G Dietz, P J M Mallens, An Integrated, Business-Oriented Perspective on Facts and Rules, data2knowledge newsletter, January-May 2001

Eriksson and Penker 2000
Business Modeling with UML: Business Patterns at Work. Wiley Computer Publishing, 2000

Habermas 1981
J Habermas, Theorie des Kommunikatives Handelns, Erster Band, Suhrkamp Verlag, Frankfurt am Main, 1981

Reijswoud and Dietz 1999
V E van Reijswoud, J L G Dietz, DEMO Modelling Handbook Volume 1, Delft University of Technology – Department of Information Systems, Version 2.0, May 1999

Searle 1969
J R Searle, Speech Acts, an Essay in the Philosophy of Language, Cambridge University Press, Cambridge MA, 1969

Gerke Geurts
gerke.geurts@logicacmg.com
Gerke Geurts is a technical architect within LogicaCMG. He has a keen interest in the many facets of software development in an enterprise setting. His waking hours are spent coaching and trying to stay ahead of clients, software developers and his children.

Adrie Geelhoed
adrie.geelhoed@logicacmg.com
Adrie Geelhoed is an enterprise architect working for LogicaCMG Financial Services and provides business-centric architecture and software development guidance to customers and consultants.

Metadata-driven Application Design and Development

By Kevin S Perera, Temenos

In this article, I present an overview of a pragmatic approach to using a metadata-driven approach to *designing* applications that has a practical presence in the *development* of applications. The article is presented in an open style and hence is not targeted at the deep technical community. I hope this will demonstrate why metadata-driven design is one of the most powerful tools at the disposal of IT professionals who are engaged on building a software product.

Remember 30?

Turning thirty was a brief affair. No run up to a massive 'fair well twenties' party, no late night at a nightclub, and no enormous hangover in the morning. But it was no less memorable for the lack of pomp and ceremony. In place of a hangover was a small realisation that formulating patterns is important to *anyone* working with technology.

I was in Singapore, to assist a small team running a package selection and integration project for a bank. I had arrived shortly before my birthday, with the intention to lay low, get some reading done, maybe to do a little work, and certainly to defer all celebrations until I returned home. A colleague of mine, Lyn, was clearly appalled by my attitude and had an altogether different idea. Post dinner on that ceremonious day I was duly press-ganged up to the seventieth floor bar of our hotel. So there we were sat, with the wonderful views out over the Singapore harbour, me drinking the ubiquitous Singapore Sling and Lyn drinking a fruit cocktail as she was nearly six months into her second pregnancy. Not a scenario I ever imagined for my thirtieth birthday. It was the first time any of us had worked closely together and hence we sat discussing the path of professional fortune that had brought us to this project. Lyn had formerly worked for SWIFT and held much of the business knowledge the project would require. As for me, my previous engagement had been on a large custom application project but this time my role was to define an architecture in which to position and integrate the various products to be employed on the project.

Clearly, Lyn and I were different characters with very different professional backgrounds. Business verses techie on the seventieth floor. The stage was set for a showdown, a tête-à-tête to the bone, but strangely we didn't end up trying to kill each other. Business verses techie it may have been, but Lyn and I reached an agreement that evening on one thing that has influenced my professional work since I first started reading the patterns literature. When it comes to IT projects, they work on patterns. And these patterns are *not* just for techies.

How high are your patterns?

Patterns of analysis, design, and implementation are virtually ubiquitous in the IT industry today and have received much publicity. Given the corpus of work freely available in the patterns area, there can be little doubt that the majority of key players in the IT industry believe applying patterns to the building of applications is a valuable and powerful approach. Many aspects of patterns feed directly into the common goals of methodologists, analysts, designers, implementers, testers and management, namely those of higher quality, more maintainable, more reliable and faster to develop solutions.

Powerful as patterns can be, they have one aspect in application design and development that has arguably proven difficult to achieve. That aspect is a substantial improvement in traceability (and hence consistency), from the *analysis of the solution domain* through to the creation of an application based on a standard framework-based implementation.

It's a simple but lofty goal; to make software quicker to develop and more reliable the IT industry needs to learn to reuse software infrastructure more effectively; to reuse infrastructure more effectively, we need to define the allowable (or supportable?) patterns of use for the target infrastructure. Introduce a more complete traceability in the software lifecycle and you get the potential to realise this, because traceability provides a *formalised* path to transform analysis-time artefacts right through to build-time artefacts. This is the key to being able to *forward-engineer* the analysis; you need to know the technical (infrastructure) environment you are planning to target with generated code.

Looking at the various infrastructure initiatives in the IT industry, across many technologies, it is clear that many software infrastructure providers are looking to *productise* the concepts around patterns in to a *framework for the general implementation of applications*. To a certain extent, this has been going on in the IT industry for many years. However, this time the infrastructure providers are looking

> "Traceability provides a formalised path to transform analysis-time artefacts right through to build-time artefacts."

much higher in the layers of the application architecture. This time, they are out to steal the wind of the application architect and put it in the sails of their framework.

Looking at what is on offer today and what is clearly in the pipeline, application architects out there should be tracking this carefully; this time we have infrastructure products *and* tools that have been creeping up towards *the base of the functional layers*. And with this creeping comes a new adversary for the not-invented-here syndrome. The tools are not just helping you to model, but are helping you to speed your coding via systematic code generation.

Want some of this? Then you need to start working with patterns, both on the infrastructure and application layers.

This *movement* in the IT industry has been gathering strength for many years. A simplified view *could* be, naturally, simple.

– The higher the patterns of [reuse of] infrastructure climb, the more we can produce repeatable, higher quality and faster time to market business applications.

The less simple, more technical view on this *should* be a more long-term perspective.

– Understand the patterns and we can build the supporting infrastructure.
– Productise the infrastructure and we can build high-productivity tools to support that infrastructure.
– Give the technical community the tools, and they will create more repeatable, higher quality applications with a faster time to market than is typical today.

Patterns, patterns, everywhere
If our work in information technology is as heavily linked to patterns as the industry appears to believe, then working with patterns is a reality for most of us. We probably don't realise the patterns we employ in our everyday roles, but this is the nature of a pervasive subject matter. If something is everywhere, after only a brief period most of us simply don't see it anymore; especially if it's related to tasks we know and perform frequently.

If they are everywhere, what does it mean to work with patterns? Many things to many people I am sure, but for the purposes of this article it has two concrete meanings. In the design process, using patterns means basing the design on commonly occurring structures ('patterns of design')[1], preferably using design tools capable of storing the design in some form of model representation such as UML. In the build process – where the real code is cut – using patterns means using software infrastructure products and tools including code frameworks and code generators such as Codesmith[2].

Fortunately, the nature of architecture and design proffers the opportunity to identify and classify patterns. The valuable work done by other authors means I don't need to engage in a long discourse on the value proposition of patterns in design. However, the same statement cannot be made carte blanche for the implementation phase of the software lifecycle where patterns – or more specifically frameworks – unfortunately have gained the occasional bad reputation.

Effectively, frameworks suffer from 'generics'. It is often possible to apply the use – or abuse – of a framework in many different ways. In providing for a reasonable number of alternative uses and styles of use, to provide a generic base service to those people building applications, frameworks have gained an unjust reputation of being overly complex and hence difficult to understand as IT professionals are swamped with information on the various approaches employed. Getting the level of a framework right, achieving the balance between prescription and flexibility, has proven a difficult challenge to meet. Get it wrong and you always face the question, do you need patterns to implement software-based systems?

The root cause of this issue is probably that most application teams are under pressure to produce higher quality solutions as fast as possible. Naturally, this leads teams away from approaches that are potentially more complex and that hence may delay the onset of the application's build. When working under pressure, nobody is looking for an approach that they feel has the potential to bring more stress.

Anyone for some stress reduction?
Thinking about stress brings a question to mind. Ever found a pattern to help relieve some stress? I have one that is absolutely fantastic. Go home, hide in the study away from the madding crowd, scan read the remainder of the days' emails and

[1] For example, as per the definitions documented so very successfully in the 'gang of four' book 'Patterns of Software Design'.

[2] http://www.ericjsmith.net/codesmith/

try to update the little work and play mind-map I keep hidden on my laptop. All the while sipping a good single malt or maybe a white port. Now that is a truly great pattern.

I have always found patterns in many places, from the manner in which start of day status meetings are run to the end of the day wind-down. It is inherent in the nature of patterns that they help us to manage complexity. A good pattern can provide a good understandable description of complex structures that can equally assist in the description of complex relations. This can make life so much more pleasant and stress-free for anyone whom needs to work with complexity. Admittedly, doing so also has a couple of other useful side-effects, but we will investigate those later in the article.

If the use of software frameworks and infrastructure can be defined by patterns [of use], then metadata is the language used to describe those patterns. Hence, describing the patterns is the *responsibility* of metadata. Metadata is not a secular language, many different dialects of metadata exist and may even cohabit. Metadata is the common term for the representation of the data models that describe patterns. Hence, for any given set of metadata the quality of the description it offers is determined by the associated metadata model (literally, the model of the metadata).

It is worth noting at this point that patterns are not and should never be restricted to design-time. Previously, standards such as the Unified Modelling Language and initiatives such as Model Driven Architecture indicated to many that modelling and metadata were upstream in the project lifecycle. Not so. IBM, like many large technology organisations, started work on formalising a model of a *total system* based in UML during the late 90's. Many of these initiatives never became official external publications, but in October 2003 Microsoft broke this apparent silence and published details of its pending System Definition Model (SDM). With models such as SDM covering the entire scope of a system, you can expect to see model-driven and hence metadata-driven processes encroaching on almost every area of the software development lifecycle.

Describe your patterns to reap the benefits of functional, policy and service abstractions

Using metadata in the design and build processes – or metadata-driven design and development of applications – proffers one solution that can elegantly leverage the powerful nature of patterns, and hence leverage the high value software framework and infrastructure products that are now available in the market.

Metadata helps to describe the patterns present in your application's domain. Describing (or *modelling*) the patterns that the application must employ helps to promote understanding of what *features* the software infrastructure must provide and the *style* in which that infrastructure could best be utilised. This provides very early visibility of the infrastructure needs and also permits, very importantly, a safer, more controlled environment in which the software infrastructure can *climb higher* in the layers of an application's architecture. Control over how infrastructure creeps up the layers is critical to providing higher value services to the application developers. This provides your applications developers with an increased level of functional abstraction.

Modelling the *needs* of an application using patterns (in turn described by metadata) may be applied to almost any area. Hence, metadata can be used to describe the mapping of object attributes to relational tables, through the definition of rules around the runtime session management of a pool of related objects to the signatures of the services offered by a particular class of object. Such rules form the policy definitions used by application developers to control the underlying behaviour of the infrastructure.

When carefully and judiciously applied to the modelling of your application, metadata may be employed in the description of both the infrastructure and application service boundaries and for the configuration data of the associated service implementations. Subsequently, the metadata descriptions associated with these services (and the metadata model) may be used in the forward-engineering of your services – into existing and future technology environments. This provides a rich – and reusable – approach to services definitions.

Using metadata in this manner, across the business (application) and technical (infrastructure) domains, requires an agreement on the overall metadata-model between the business analysis and the software infrastructure. This is an agreement on a *behavioural contract* between the application and its infrastructure and permits both the detailed functional analysis and the

> "The metadata concept may be applied to many areas of technology, and hence to many software products, in the business and technical layers."

construction of software infrastructure to run in parallel. In such a scenario, the application development team may now tackle critical issues at a very early stage in the development process which is known to be a key factor in improving the effectiveness of the software development lifecycle.

The metadata-driven approach also works well with other mainstream approaches such as use case modelling and artefact-based methodologies (for example, the Rational Unified Process), has the potential to naturally lend itself to reuse and repeatability, and is independent of the target technology platform.

Technology

Before diving into the discussion about how metadata links to the Service Oriented Architecture and Web Services, it is probably worth a little pause at this point to discuss the technology behind metadata. In thinking how should this be best formulated, I came to the conclusion that there is no easy way to say this in a technology-related article. So let's just put it out there – quite simply, there is no 'technology' in metadata.

Metadata itself is a concept. That is why many different models of metadata – or metadata models – can exist. The metadata concept may be applied to many areas of technology, and hence to many software products, in the business and technical layers. JNDI has a meta model, as does LDAP and Active Directory. They all use similar concepts in their meta models, but all have different (physical) metadata. What is clear is those design and development products using metadata also use (embed) or link (integrate) to tools. Frequently these tools are graphical and, more and more, are linked to improving development productivity. Next to the metadata itself, which clearly has a primary importance, this use and integration of tools to support the storage, propagation and use of your metadata model across the software lifecycle is probably the most important aspect of technology in the design and build environments.

When selecting tools to assist with design and development, the most important criteria are often centred on:

- the ability of the tools to store and use metadata, both of their design and yours.
- the ability of the tools to share metadata, which is particularly useful between the design-time and build-time to help reduce the semantic gap between the design and the implementation.
- the integration of the development tooling with the underlying software infrastructure and runtime platform, particularly for code generation.
- the integration of the design and development tooling with code generation technology, such as scripting languages that may be used to interrogate the metadata model.
- standards (MOF) compliance.

Linking SOA and Web Services using metadata

Instead of creating a several thousand word essay on why using metadata in design and development is so great, I'll try an alternative, more 'techie-friendly' approach. Let's assume we all agree that metadata-driven design and development is just fine. Patterns are great, patterns govern most of what we do in IT, metadata is a cool way to represent your patterns, and finally that metadata can be used in both design and build.

As we all agree that metadata-driven design and development is great, we don't have to waste any more time debating its merits. In place of the essay, let's take a look at what we could implement with this approach based on a link between SOA and Web Services.

First, a cautionary note to the reader before continuing. The examples are exactly that – examples. Focussing on one aspect of using metadata in a system does not mean the same concepts do not apply to other areas. Many areas of systems already use accepted metadata models, particularly around authentication and authorisation using products such as Active Directory/LDAP and certificate management services.

The proposition

Walking through two example applications built on similar principles but different implementations, I am going to explore how a metadata-based approach can be used to define the service interfaces on an existing application and on a new application. The primary goal of this exercise is to propagate those interfaces to an additional technology environment using a combination of infrastructure products, a (code) framework for building applications (an application framework) and a metadata model seeded in the design phase.

Example applications – common ground

In this walkthrough, the two example applications perform similar functional roles and have a similar façade to the exterior world.

As I am focussing on the service interface layer on the core server, I'm going to make a few assumptions about the implementation of each application. First, both applications are running on the same database and data model – application #2 is the candidate replacement for application #1. Second, I am not concerned with the presentation interface or any middle-tier user interface components. Third, I am not concerned with any detailed deployment aspects or configuration of any communications mechanisms. Fourth, the server model is identical, with application sessions shared across all requests for services and server-side session handling embedded in the application or the runtime platform. Fifth, and last, the development of the application is carried out by multiple teams in several time zones.

While the actual (business) functionality may be similar between the applications, what is not common between the applications is the model for presenting their service interfaces on the system boundary.

Example application #1 – the original, the legacy

The server-side interface of the original application was comprised of a C header file. Each function on the interface employed a signature tailored to its purpose. External components called to the interface functions using a DCE-based remote procedure call.

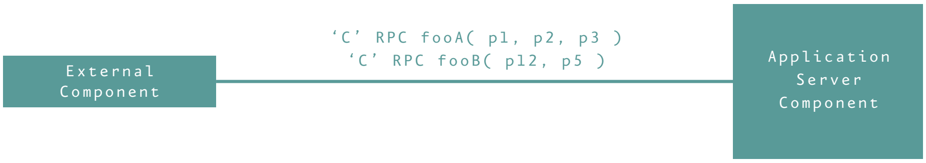
Adding a new function is relatively simple, but highly coupled to the DCE technology. Adding an alterative access to provide a non-DCE invocation channel to the function is not possible without significant re-engineering.

Note that the data dictionaries of all services are exposed directly on the application's interfaces and that function names are placed directly in the global namespace of the application (they must be unique across all functions being developed by the teams). This is shown in *Figure 1*.

Example application #1 – the evolution

The original application #1 was later wrapped by a set of C++ classes to expose the server-side interfaces on a CORBA bus. The C++ interfaces are not direct wrappers, but group 'C' functions in to sets of services and provide a uniform 'generic' object as the request context.

Figure1. Application #1



Each service defines a number of classes derived from the request context class, to create a set of dedicated 'containers' for the parameters required by the underlying 'C' functions on the server-side. Thus functions are grouped into services and the parameter definitions for all functions are encapsulated by the set of container class definitions. To facilitate the handling of these 'more generic' requests, a new server-side (application) component type was introduced to check the request context to validate the correct parameters have been supplied and map these parameters to the correct underlying 'C' function. The request mapper is the implementation of the service defined in the interface definition language (IDL). Normally, to permit parallel development, there is one request mapper per service IDL. However, this model is not enforced by the system.

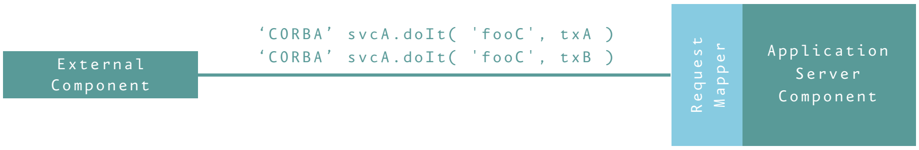
Adding a new function to an existing service requires the derivation of a new container class and the addition of new code in the mapper to marshal the container data and make the underlying 'C' call. Adding a new function to a new service requires first that the service interface be defined in IDL and a mapper be created based on that service interface.

Note that the data dictionaries of all services are exposed directly on the application's interfaces, but are scoped by service and request context definitions as shown in *Figure 2*.

Example application #2 – the King of the Hill?

The key difference between application #2 and application #1 is that application #2 was built with a

Figure 2. Application #1 evolved

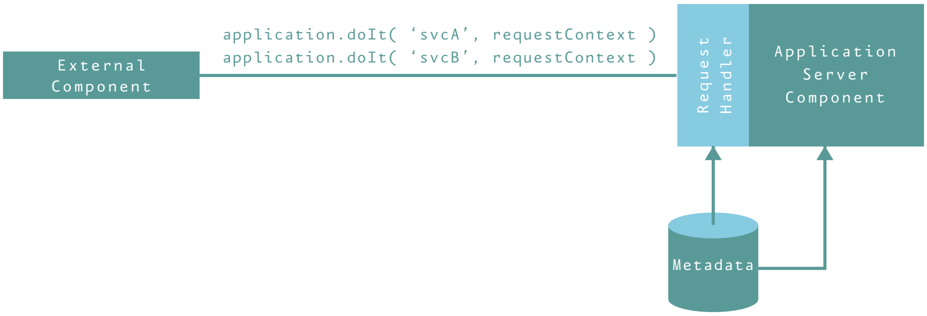

'CORBA' svcA.doIt('fooC', txA)
'CORBA' svcA.doIt('fooC', txB)
External Component
Request Mapper
Application Server Component

metadata-based approach. Application #2 is not restricted to reusing the existing 'C' functions.

Each server-side function in application #2 is written based on a common metadata model for the representation of a service request. This is similar to the evolved application #1 in concept, but the handling of the generic request contexts is embedded in the server code. Server-side functions must now understand the generic request context format. This is shown in *Figure 3*.

To represent the request context, concrete parameters are not desirable. In place of a concrete function signature, at runtime application #2 is using a generic structure for the request context. However, in this instance the request context needs to be similar in nature to a name-value pair property-bag. Property bags can be hierarchical and hence can be used to represent any arbitrary data structure much like an XML document based on an XML Schema.

Figure 3. Application #2


application.doIt('svcA', requestContext)
application.doIt('svcB', requestContext)
External Component
Request Handler
Application Server Component
Metadata

The service name and request context are all that is required by the new server-side application components to execute the request. The service name coupled to the definition of the permitted structure of the request context is persisted in the database as a 'service definition'. This definition, effectively an entry (row) in the database, belongs to that service alone.

A new generic request handler application module is included in the system. This module has been designed to interrogate the metadata repository and validate incoming requests against the associated service description metadata. It can also create the internal memory structure from the request context to pass to the target service implementation.

Thus the interface definition is held in the metadata repository (in this case, the applications database).

Adding a new service to the application is straightforward on the server-side. Implement the generic interface and manage the request context structure required for the service request, assign the service a name, register the name and request context description in the database and perform any deployment actions required to deploy the new service code on the server.

The same cannot be stated for the client side. In place of the definitive client-side header file or IDL interface definitions of application #1, the client-side of application #2 receives only a generic entry point function via the same technical-interfacing technique. This is a big change, in that the service data dictionaries are no longer propagated directly onto the service interfaces. The service interfaces, from an external perspective, are all soft-set in the database. Note also that the scope of a service definition is now one-to-one with the service implementation.

Clearly, application #2 suffers from an imbalance. The server-side is now very flexible and well defined, while the client-side needs to understand the metadata before it is possible to understand what services are on offer.

Enter the world of service discovery. The services of application #2 need to be discoverable by clients wishing to use them. This is possible via the tried and trusted pattern called *documentation*, Users (clients) of the applications services can read the interface specification and therefore construct well formed and validate requests.

Metadata-driven SOA

Obviously, example application #2 is very service oriented at the system boundary – it does not know anything else – but it would be unfair to claim example application #1 does not embody some aspects of a service oriented architecture.

In its original form, application #1 is like many so-called legacy applications written in 'C' or COBOL. The code may have been modified many times, but within the domains of the technologies available at the time these applications were fundamentally service-oriented. 'C' programs had their extern function signatures; COBOL programs had their copy books. Include an infrastructure like IMS or CICS, and you certainly have a service-oriented system façade as these TP monitors forced that model – a bucket of data in, run the request, a bucket of data out and describe the data buckets via data structures. That is a service, right?

So what is the fundamental difference between applications #1 and #2? The answer [for me] is they are both service-oriented at their system boundaries, but to be a true service-oriented application the fractal model must be applicable from the system boundary to the database, with service interfaces defined for each component or sub-system and each service treated as a black-box by the caller.

However, staying concentrated on the system boundaries of the examples indicates that the quality of the service description is vastly different between them.

- The original application #1 is doing nothing more than exposing library or executable end points, with its entry point function signatures propagated over the system boundary via the DCE technology.
- The evolved application #1 is doing the same, but with a more elegant end point description mechanism employing a simple abstraction. Entry point functions are exposed as a simple service-oriented abstraction via CORBA, with the link between the abstraction and the concrete implementation via coded marshalling.
- Application #2 has only one technical end point to expose. Exposing this end point in any technical environment in the same manner (via DCE, CORBA, COM, asynchronous messaging, etc.) will generate the same result – the data dictionary will never be exposed, only the signature of a generic service call. The value in this approach is to use the metadata-driven service definition and the associated application infrastructure for invoking those services, as the implementation runtime for generating many different *proxies* in a number of different technologies to permit wider usage of the services.

The metadata-driven nature of the services of application #2 leads the solution to a dead-end if a pure technical 'code it' approach is taken to providing access to those services. In such a metadata-driven application exposing functions is replaced by exposing metadata.

Sound a little dangerous, this idea of exposing metadata? Exposing the metadata itself is not the true intent of a metadata-driven application. Using the metadata to drive the propagation of services [functions] over the system boundary is a more accurate manner of phasing the approach that needs to be employed.

From SOA to CORBA, from SOA to Web Services, from SOA to …

To demonstrate how a real metadata based service design can outrun the competition, let's propose some new modifications to our existing example applications.

- We have our basic server-side runtime setup for all three applications, this stays 'as is'.
- We would like to add a clean migration path to a very fashionable new integration capability based on Web Services.
- We also want a migration path for a number of the external applications that have been integrated via CORBA against a number of specific services of application #1.
- Someone asked for documentation, so let's give them that too.
- These external applications are to be migrated to use the corresponding services of application #2, requiring that application #2 supports the same type of CORBA interface as employed in the evolved application #1.

Providing support for both CORBA and Web Services access channels to the services of application #2 may all be achieved efficiently, effectively and safely using an infrastructure and code generation. The same cannot be stated for either guise of application #1. If the same approach were to be applied to application #1, the most likely result would be the sufferance of increased complexity from the lack of a central model to describe its services. Note it is the central model that is important, much like the TModel

"… a bucket of data in, run the request, a bucket of data out and describe the data buckets via data structures. That is a service, right?"

concept in UDDI being the central model (as opposed to the physical data model used in MSDE or SQLServer as the repository).

Using the metadata service definitions coupled to an appropriate tools and infrastructure environment, a metadata-driven application is capable of providing this *bridging* approach to propagate its services in to many technologies via code generation. This is a direct result of all services being regular (described in a common grammar and vocabulary) and that all service descriptions are available in a meta-format (a descriptive format – the metadata) at both build-time and runtime.

Design direct to implementation – linking metadata to tools
There are a number of prerequisites to this approach that are formidable enough subjects to warrant articles of their own. But to touch on them briefly, in a horribly simplified vein, we need to go shopping for the following items.

- Modelling tools that are capable of providing a graphical interface on to a model repository where we may store our metadata models and the associated metadata instances (not necessarily in the same physical store). Dependent on your application domain the usual suspects are likely to be an UML-style tool, a process modelling tool or a combination of the two. The tools must provide a programmatic interface to their model storage.
- One or more target infrastructure products, preferably standards based or de facto third party standards based but home-grown if required. These products should be focussed on providing the bulk of the technical services required by an application, from persistent storage to session management (and that is session management on all tiers of the application, where possible).
- Development environment tools capable of supporting the programmatic interfaces of the modelling tools and the target infrastructure products.
- Development environment tools capable of supporting code generation. For those familiar with Lex and Yacc, this is not the type of 'tool' in question. Supporting code generation in this context needs to be at a higher level than the basis of regular expressions and context-free grammars. Ideally, code generation tools should embody some of the principles of dealing with metadata as this helps significantly reduce the complexity of the code generator.
- One or two strong technical leaders, familiar with the concept and use of employing metadata to drive the application development lifecycle.
- A number of designers familiar with the concept of metadata modelling and the design of application infrastructure.
- A number of developers familiar with the concept of using metadata and whom agree with the designers.

The modelling tools, infrastructure products and development environments all exist in today's marketplace. The final 'people-oriented points' are arguably the most difficult prerequisites to get right, as is the case with most software teams it is the mix of people and approaches that often makes or breaks the software lifecycle. Professor Belbin's test will help get the mix of 'people types' reasonably well balanced, but there is no compensating for a unified team[3]. This is critically important in a metadata-driven approach, as all team members must adhere to the model if the application is to achieve its goals.

Putting the whole thing together, we arrive at a process whereby it is viable for the solution analysis (sometimes termed the requirements analysis) to feed directly in to the application design and the application infrastructure. The application design governs the overall schema of the component model and the metadata definitions. The application infrastructure looks to provide support for the application schema via a set of application framework APIs, the reuse of standard infrastructure or building of bespoke infrastructure. Finally, the application tooling is responsible for interrogating the models and metadata of the application's design and generating code on top of the application infrastructure. This is shown in *Figure 4*.

Build-time
Looking a little more closely at the build-time, and once again focussing on the service interfaces, the metadata service definitions are used by the application tooling to provide code generation of the service interface implementations (the service proxies, in different technology environments).

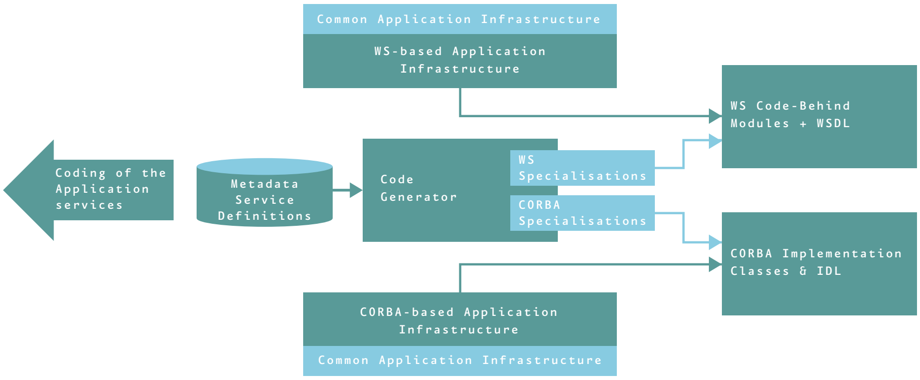
The application's metadata is used directly to derive generated code. It is important to note the metadata feeds in to the build of the application's services also, as does the common infrastructure of the application. The generated code

[3] The potential impact on team dynamics and corresponding management techniques when applying these techniques is outside the scope of this paper and is a substantial enough subject in its own right to warrant a dedicated article.

should be based on the same code base (the framework) as the core application. This helps to keep the volume of generated code to a minimum – code is generated on top of this infrastructure layer. This is important to help reduce the need for complete regression testing.

The role of the generated code is to providing the marshalling of requests from one format (such as a request originated from a CORBA peer, or a Web Services POST) to the generic internal metadata-oriented request format. The metadata must contain information about the service interface, such as parameters types and names, but can be extended to include default values and validation. If extended in this vein, the generated proxy will be able to use the same validation rules as the core service (on the assumption that the same metadata is used by the core service – which in this scenario, it should be!).

Figure 5. Systemic code generation


Common Application Infrastructure
WS-based Application Infrastructure
WS Code-Behind Modules + WSDL
Coding of the Application services
Metadata Service Definitions
Code Generator
WS Specialisations
CORBA Specialisations
CORBA Implementation Classes & IDL
CORBA-based Application Infrastructure
Common Application Infrastructure

The generic infrastructure is, fundamentally, extended to support a specific technology via systemic code generation. The code generation should simply be defining a wrapper on the underlying application infrastructure to marshal requests from 'one side to the other'. This is made possible by the presence of a metadata model, as that model defines an overall structural representation for the service interfaces. The result is that the infrastructure for a given technical environment can be tested independently of the code generation, a key factor in increasing the quality of a solution employing code generation. Deriving from the metadata also permits different code generation policies to be applied, such as 'include code generation for request validation' verses 'no code generation for validation'.

One downside to code generation is the need to ensure that all critical requirements of the target environment (to be bridged by the code generator) are well described in the service metadata model. If not, the metadata model needs to be revised or extended to support the concepts in the target environment that are absent from the metadata model (that is, the model is not complete).

Another downside to deriving the service proxies in this manner is that of versioning. Versioning and configuration management is often a very thorny subject and certainly a subject that warrants dedicated treatment. However, the issues facing proxy versioning are not so different from the issues faced by more typical development approaches. If a service definition changes, the associated proxy will need to be regenerated and any integration against that proxy will need to be assessed for impact (all facilitated by the formalised traceability of the approach). This is no different to any other scenario, except here we need to ensure the core services, infrastructure and generated proxies are all matching what was intended in a deployment!

For the former, there is no real answer to this as it is all about the completeness of the metadata model being utilised. Looking for standards may help, but nothing will beat a well reviewed and documented (use cases!) model. Fortunately, for the latter, derivation from a metadata model means both packaging and impact analysis on changes to packages or packaging may be performed within the same tools-based regime. At code generation, the exact configuration of the build is well known and can be used to populate a deployment repository.

Run-time

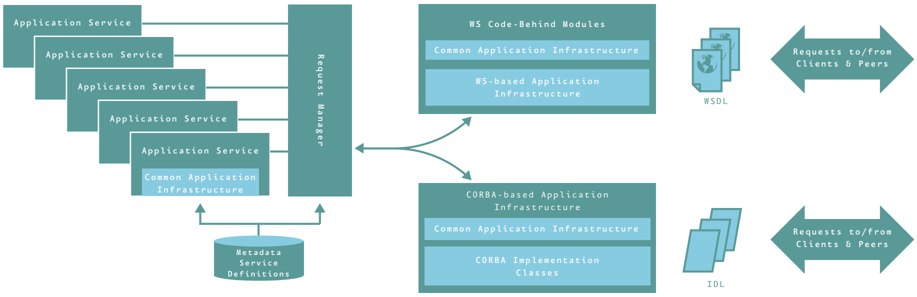
Returning to the original intent of showing what can be done with a metadata-driven approach to design and development of applications, the best place to review that is in the runtime.

Assuming the build process used a common application infrastructure tailored towards the metadata service definitions and that metadata was used in the creation (coding – no magic there!) of the core services of the application as well as the generation of service proxies, the consistency of the overall solution will be as complete as the metadata model itself.

A request originating from any supported external source is marshalled by the respective service proxy. In that marshalling, dependent on the code generation policy employed, the service proxy may perform some validation of the request based on the rules supplied via the metadata. This could even include authentication, authorisation and session management via a hand-off of metadata to delegated sub-systems.

Once marshalled in to the generic service request format the request is forwarded to the request manager for execution. In that execution, the request manager interrogates the request and matches it against the metadata for the target [core] service. If all is well, the request is accepted by the target service and that service may then use additional metadata in the processing of the request. This is shown in *Figure 6*.

Figure 6. Request processing


Application Service
Application Service
Application Service
Application Service
Application Service
Common Application Infrastructure
Request Manager
Metadata Service Definitions
WS Code-Behind Modules
Common Application Infrastructure
WS-based Application Infrastructure
WSDL
Requests to/from Clients & Peers
CORBA-based Application Infrastructure
Common Application Infrastructure
CORBA Implementation Classes
IDL
Requests to/from Clients & Peers

The final result is an approach to solution design that flows from inception to implementation, providing a pragmatic view on how and why metadata-driven applications potentially have a longer life than their more traditional counterparts.

Documentation

If this approach permits the generation of code from service definitions, there is no reason not to generate the service interface specification documentation also. This has been common practice for many years, with top-down documentation generation from tools like Rational Rose (to Microsoft Office Word) or bottom-up via code-oriented tools such as AutoDuck or JavaDoc.

Applicability

Worthy of a few observations, is a brief summation on the organisational aspects involved in the application of this approach. To characterise the types of organisations that often appear drawn to a metadata-driven philosophy to developments, this approach probably is geared:

- for designers and developers that want a closer link between the design and the implementation;
- for organisations that are looking seriously at a real infrastructure-driven, higher productivity and higher quality approach to their development;
- for organisations that are not nervous about getting in to bed with a couple of productivity enhancing tools;
- for organisations looking to build knowledge repositories and the associated tools, to better describe their product(s) via a formalised description language;
- for organisations whom need to support a range of technical environments, particularly at the system boundary.

Positive benefits?

This approach has been used to varying degrees on many projects I have discussed, reviewed and worked with. Many of these projects have sought the high goal of complete forward engineering from the models to the runtime, and some have found success when dealing with a specific domain.

At Temenos, we have been using this and related techniques in the production of the new 24x7 capable banking system, 'T24', launched late in 2003. More specifically, we have used these techniques in the production of a software development kit (programmatic APIs on existing functions) and Web Services deployment tooling. It might not be easy, and it might hurt your head from time to time. But looking at the model for the T24 solution it is clear to us that a metadata-driven approach to the design and development of your applications will, when the next technology wave comes, help you engineer your existing services out of a hole and in to the limelight. Check, carefully, the initiatives of many of the IDE and tools vendors. Metadata representation and code generation is being courted once more. This time however, the aim appears to be to help the development process become more productive by providing tools to manage the abstractions and complexity in today's technical environment.

References

Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects
Douglas C Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, John Wiley & Sons

Anti Patterns
William J Brown, Raphael C Malveau, Hays W McCormick III, Thomas J Mowbray John Wiley & Sons

Applied Microsoft .NET Framework Programming
Jeffrey Richter, Microsoft Press

Application Architecture for .NET, Microsoft Patterns & Practices Group
Microsoft Press

Beyond the Component Based Process
CBDi Newswire Commentary
www.CBDiForum.com

Design Patterns
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley

Dinosaurs Battle in the Tar Pits
CBDi Newswire Commentary
www.CBDiForum.com

Modelling Languages for Distributed Applications
White Paper, Microsoft Corporation

Modern C++ Design
Andrei Alexandrescu, Addison-Wesley

Objects, Components and Frameworks with UML
Desmond Francis D'Souza & Alan Cameron Wills, Addison-Wesley

SOA – A Cost Reduction Strategy
CBDi Newswire Commentary
www.CBDiForum.com

Service Oriented Process Matters
CBDi Newswire Commentary
www.CBDiForum.com

Structure and Interpretation of Computer Programs
Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press

Kevin S Perera
kperera@temenos.com
kevin@sandokan.co.uk

Working within the Technology & Research team at TEMENOS, a core systems provider to the finance industry, Kevin Perera is engaged in the role of Systems Architect. Advising and guiding the key design and development teams of the company, Kevin's primary goal is to ensure the overall consistency of the solution components and their alignment with the technical strategy of the TEMENOS product suite. His main areas of focus during 2003 have been XML interfaces and protocols, with a particular interest in the provision of APIs for 'business services' provided by the existing applications of the company in new technology domains such as .NET and Web Services.

Best Practices for Rule-Based Application Development

By Dennis Merritt, Amzi! Inc.

The word 'knowledge', like many words adapted for computer science, has a technical meaning that is different from its common meaning – and like many such words, it has been defined and re-defined many times to suit the needs of various trends in computer science.

This paper takes a high level view of knowledge, using the word in its more general sense, rather than as a specific technical term, and then looks at different types of knowledge and their mappings to executable computer code. The purpose is to gain insights into when and why rule engines provide advantages over conventional software development tools.

The three types of knowledge considered are *factual, procedural,* and *logical*. These divisions correspond to the capabilities of computers. The first two map naturally to a computer's architecture; the third does not.

Factual Knowledge

Factual knowledge is just that, facts, or data. It can be facts about customers, orders, products, or the speed of light.

Computers have memory and external storage devices. These are ideally suited to the storage and retrieval of factual knowledge. Database tools and programming languages that manipulate memory have evolved naturally from these basic components of machine architecture.

Factual knowledge appears in the computer as either elements in a database or variables and constants in computer programs, as shown in *Figure 1*.

Procedural Knowledge

Procedural knowledge is the knowledge about how to perform some task. It can be how to process an order, search the Web, or calculate a Fourier transform.

Computers have a central processing unit (CPU) that processes instructions one at a time. This makes a computer well-suited to storing and executing procedures. Programming languages that make it easy to encode and execute procedural knowledge, have evolved naturally from this basic computational component.

Procedural knowledge appears in a computer as sequences of statements in programming languages, as shown in *Figure 2*.

Logical Knowledge

Logical knowledge is the knowledge of relationships between entities. It can relate a price and market considerations, a product and its components, symptoms and a diagnosis, or the relationships between various tasks.

Unlike for factual and procedural knowledge, there is no core architectural component of a computer that is well suited to the storage and use of logical knowledge.

Typically, there are many independent chunks of logical knowledge that are too complex to store in a database, and lack an implied order of execution which makes them ill-suited for programming. Because it doesn't map well to the underlying computer architecture (as shown in *Figure 3*), logical knowledge is difficult to encode and maintain using the conventional database and

Figure 1. Factual Knowledge

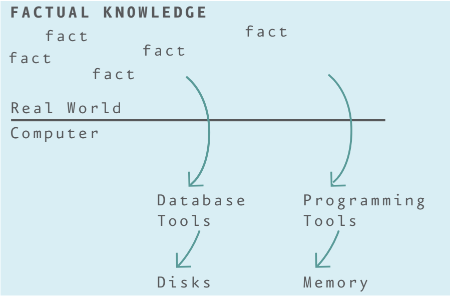


Figure 2. Procedural Knowledge

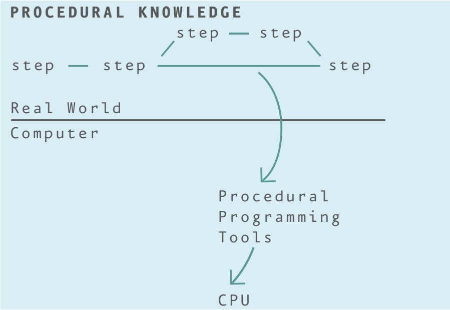


Figure 3. Logical knowledge does not map well to computer architecture

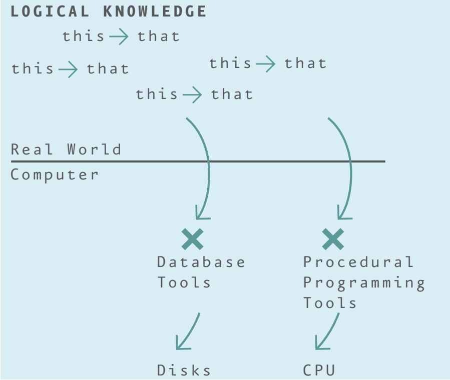
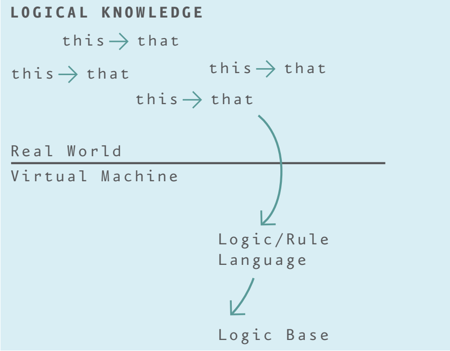


Figure 4. Using virtual machines for logical knowledge



programming tools that have evolved from a computer's architecture.

Specialized tools, which are effectively virtual machines better suited to logical knowledge, can often be used instead of conventional tools (as shown in *Figure 4*). Rule engines and logic engines are two examples.

Conventional vs Specialized Tools

Logical knowledge is often at the core of business automation, and often is associated with the 'difficult' modules of an application. Consider, for example, a pricing module for phone calls or airline seats, or an order configuration module. Furthermore, logical knowledge is often changing. Government regulations are expressed as logical knowledge, as are the effects of changing market conditions. Business rules that drive an organization are almost always expressed as logical knowledge.

Because of the critical role logical knowledge can play, there are good arguments for using specialized tools which make the encoding of logical knowledge quicker, easier and more reliable. There are also, however, good arguments against them, foremost being the ready pool of talent that is familiar with conventional tools. There is a lot to be said for sticking with the familiar, although in general the cost is lengthy development times, tedious maintenance cycles, a higher than normal error rate, and often compromises in the quality of service the application provides. On the other hand, there are some well known problems with rule engines and other tools designed for working with logical knowledge:

– There are many choices, and they are usually vendor specific. There isn't a standard rule language to use.
– Each tool is better suited for some types of logical knowledge than other types. Rules that diagnose a fault need to behave different from rules that calculate a price, which in turn behave different from rules that dictate how an order can be configured.
– Maintenance is not as easy as sometimes promised. It is important the rules all use similar terms and definitions, otherwise the interrelationships between the rules don't work as intended – making maintenance difficult. Furthermore, because there is no order to the rules, tracking interrelationships can be difficult.
– There is no standard application program interface (API) for integrating a rule engine with other components of an application.

Given these difficulties, is the payoff from using rule-based tools worth the investment? In many cases, yes! For example, one organization that provides online mortgage services replaced 5000 lines of procedural code to price mortgages, with 500 lines of logical rules. The logic-based solution was implemented in two months as opposed to the year invested in the original module. Maintenance turn around due to changing market conditions was reduced from weeks to hours, and the errors in the resulting code went to practically zero.

One reason for near zero errors is simply that there's less code to go wrong, but the main reason is the code closely reflects the logical knowledge as it is expressed. There is no tricky translation from business specification to ill-suited procedural code.

The biggest win of all might be the flexibility the logic-based solution provided them with, allowing them to expand their product offerings. They could now offer more and better mortgage pricing options for their customers, including the option of customizing the pricing logic for each institutional customer.

This is not an uncommon story. The same benefits and 10 to 1 improvement ratio appear over and over again in the success stories of vendors of rule-based and logic technologies.

Semantic Gap

The concept of 'semantic gap' can be used to explain many of the issues with logical knowledge. A semantic gap refers to the difference between the way knowledge to be encoded in an application is naturally specified and the syntax of the computer language or tool used to do the encoding. For example, you can use assembler to code scientific equations. But it is tedious and error-prone because there is a large semantic gap between the syntax of assembler and an equation. The FORTRAN scientific programming language was invented to reduce the semantic gap. It allowed a programmer to code an equation in a way that was much closer to the way a scientist might write the equation on paper. The result was easier, quicker coding of engineering and scientific applications, and fewer errors.

Factual knowledge and procedural knowledge are both readily coded in computers because there is a reasonably small semantic gap

> "Because of the critical role logical knowledge can play, there are good arguments for using specialized tools which make the encoding of logical knowledge quicker, easier and more reliable."

between the way facts and procedures are described and the tools for encoding them. As pointed out previously, this is because computers are inherently good at facts and procedures. The semantics of logical knowledge however does not map readily to conventional tools. Consider this piece of knowledge:

The price of an airfare from Cincinnati to Denver is $741 if departing and returning midweek. It's $356 if the stay includes Saturday or Sunday.

The meaning, or semantics, of this knowledge is best captured in a pattern-matching sense. It really means that the details of a proposed trip should be matched against the conditions in the rule, and the appropriate rule should be used to determine the fare.

This sort of knowledge could be shoehorned into procedural code, but the semantics of procedural code are designed to express a sequence of operations, not a pattern-matching search. On the other hand, a rule engine is designed to interpret rules in a pattern-matching sense, so rules entered in such a tool will have a smaller semantic gap than rules encoded procedurally.

There's If Then, and Then There's If Then

It is very tempting to store if-then logical relationships in procedural code, especially since procedural code has if-then statements. In fact, not only is it tempting, it can work reasonably well up to a point. However there is a big difference between a logical relationship and a procedural if-then. A procedural if-then is really a branching statement, controlling the flow of execution of a procedure. If the condition is true, control goes one way, and if not control goes a different way. It's a fork in the road.

It's the road bit that causes the trouble. A logical relationship can be coded as a procedural if-then, but must be placed somewhere along the road of execution of the procedure it is in. Furthermore, if there are more logical relationships, they too must be placed at some point in the procedural path – and, by necessity, the placement of one affects the behaviour of another. It makes a difference which rule gets placed first, and if there are branches from previous rules, and which branch a following rule is placed on.

This is not a problem if the rules map easily to a decision tree, but in that case the knowledge is really procedural. It's also not a problem if there are a small number of rules, but as the number of rules increases it becomes very difficult to maintain them as forks in a procedural flow. The arbitrarily imposed thread of execution that links the various rules becomes extremely tangled, making the code difficult to write in the first place, and very difficult to maintain. This isn't to say it can't be done, or indeed that it isn't done; it often is. However, the module with the rules is often the most troublesome module in a system.

Once encoded procedurally, logical knowledge is no longer easily accessible; that is, it no longer looks like a collection of rules and declarative relationships. The knowledge resource has, in a sense, been lost and buried in the code, just as a scientific equation can no longer be read if it is coded in assembler. The same is not true of either factual or procedural knowledge. In those cases, reading the code generally does show the underlying knowledge.

Databases for Rules

It is possible, in some cases, to shoehorn logical relationships into a database. If the relationships can be represented in a tabular form, then a database table can be used to encode the rule. So for example, if the amount of discount a customer got was dependent on the amount of previous sales at a few different levels, this could be represented as a table and stored in a database. However, as with the using procedures, the database approach is limited in that it only works for very clean sorts of logical relationships.

A Mixed Approach

Sometimes applications use a mixture of both the procedural and database approaches. Logical relationships that can be expressed in tables are stored in a database, and the remaining relationships are coded as procedural if-then statements.

This can simplify the coding task, but it makes maintenance harder because the logical knowledge is now spread across two different vehicles.
Despite these difficulties, there is a strong appeal to using data, procedure or both to encode logical knowledge, and that is that they are familiar techniques, and there are numerous individuals skilled in their use.

Artificial Intelligence

The problems with encoding logical relationships were first explored back in the 1970s by researchers at Stanford University. They were trying to build a system that advised physicians on courses of antibiotics for treating bacterial infections of the blood and meningitis. They found that the medical knowledge consists mainly of logical relationships that can be expressed as *if-then* rule.

They attempted many times to encode the knowledge using conventional tools, and failed because of the problems described previously.

If the problem with coding logical knowledge is that the nature of a computer is not well-suited to expressing logical relationships, then clearly the answer is to create a machine that is. Building specialised hardware is not very practical, but it turns out a computer is a good tool for creating virtual computers. This is what the researchers at Stanford did. They effectively created a virtual machine that was programmed using logical rules. This type of virtual machine is often called a rule engine.

Why is a computer good at building a rule engine, but not the rules themselves? It is because behaviour of a rule engine can be expressed in a procedural algorithm, along the lines of:

– Search for a rule that matches the pattern of data
– Execute that rule
– Go to top

The Stanford researchers who were working on the first rule-based systems had originally called their work 'heuristic programming,' which is, of course, a fancy way of saying rule-based programming. Because a large amount of human thought seems to involve the dynamic applying of pattern-matching rules stored in our brains, the idea surfaced that this was somehow 'artificial intelligence'. However the real reason for the growth of the term was pure and simple marketing – it was easier to get Department of Defence funding for advanced research on Artificial Intelligence (AI) than it was for heuristic programming. The term 'expert system' was also invented at about this time for the same reasons.

The media too, was very excited about the idea of Artificial Intelligence and expert systems, and the software industry went through a cycle of tremendous hype about AI, followed by disillusionment as the technology simply couldn't live up to the hype. Those companies that survived and continue to market and sell the technology have found the term AI to be a detriment, so they looked for a different term. Now it is most often called rule-based programming.

Whether you call it heuristic programming, Artificial Intelligence, expert systems, or business rule processing, the underlying technology is the same – a virtual engine that uses pattern-matching search to find and apply the right logical knowledge at the right time.

Other Logical Virtual Engines
Virtual engines programmed with declarative rules are not the only example of specialized software designed to deal with logical knowledge. Other such programs dramatically altered the course of the history of computing.
In the early days of data processing, reports from a database had to be coded using COBOL. But reporting requirements were specified as logical relationships–these columns, these partial sums, etc. Culprit was a report writer that let a user specify in a declarative, logical, way the knowledge about a report. It would then generate the procedural code to make the report happen.

The benefits were exactly as we've discussed – users could now create and maintain their own reports without having to go through a programmer. The result was quicker reports, faster turn around of new reports, and reporting that met user's needs much better than procedural approaches channelled through programming groups.

The resistance to this technology was also exactly the same. Data processing departments did not want to use a separate tool for reports, they knew COBOL. The product only became a commercial success when it was marketed to the end-users, and not data processing departments.

Culprit was the first commercial software product from a non-hardware company, launching the software industry.

The VisiCalc spreadsheet program was another example. It let users easily describe the logical relationships between cells without having to write procedural code. As with rule-based languages and report writers, the key was a virtual engine that translated the logical knowledge into executable procedural code.

Spreadsheet applications drove the early acceptance of personal computers.

"If the problem with coding logical knowledge is that the nature of a computer is not well-suited to expressing logical relationships, then clearly the answer is to create a machine that is."

Data & Process First, Then Logic

Recent work at Stanford has explored the question of why AI is not more widely spread in medicine, but their observations apply to application software in general. Logical knowledge expresses relationships between entities, and unless those entities are available for computation, the logic cannot be automated.

One can write a rule-based system that helps deal with patients and other aspects of health care, but without the underlying patient data to reason over, the logic base is of limited value.

The same is true for other application areas. Without a database of product components, you can't build a configuration system, and without the raw data of phone call dates, times and durations, you can't implement a pricing module.

The lack of underlying data reflects the delay in wide-spread adaptation of new technologies. It is just recently, for example, that a higher percentage of doctor's offices are using computer based patient records.

Likewise, it is only in recent years that more and more data is becoming available in relational databases that lend themselves to multiple application uses, rather than the application-specific files that have been used for most of the history of computing.

Early AI success stories are related to applications that dynamically gather the data from users, as in diagnostic systems, or that work with organizations that have good computerized records, such as insurance and phone companies. As more and more data becomes readily accessible for multi-application use, there will be more and more applications deploying logical knowledge.

Logical Knowledge Tools

Tools for encoding and deploying logical knowledge are relatively straightforward. The two critical parts are a *knowledge representation language* and a *reasoning engine*.

Knowledge Representation Language

Knowledge representation is the syntax of a particular tool. Each tool allows the entering of logical knowledge in a certain format, which might be simple if-then statements referencing simple entities, or complex if-then statements that reference complex objects and properties. They can be in and English like syntax or more closely resemble formal logic. The classic design tradeoffs of ease-of-use versus expressive power apply.

A tool might provide other means for expressing logical knowledge as well, such as hierarchical structures,, and might include capabilities for expressing uncertainty. Uncertainty is useful for some types of applications, like diagnosis where there isn't a clear right answer, but just gets in the way for something like pricing where there is only one right answer. Uncertainty itself comes in competing flavours – fuzzy, Bayesian, and the original 'certainty factors'.

Reasoning Engine

The reasoning engine determines how rules will be applied at runtime. All reasoning engines are basically pattern-matching search engines, looping through the rules, deciding which to use next, and then repeating the process until some end condition is reached. However, there can be major differences in how patterns are searched for, and what happens when a rule is used.

The two basic strategies are *goal driven* and *data driven*. A goal driven reasoning engine looks for a rule that provides an answer for a goal, such as price. Having found one, it then looks for any sub-goals that might be necessary, such as customer status. A data driven reasoning engine looks at the current known data and picks a rule that can do something with that data. It then fires the rule, which will add or otherwise change the known data. For example a customer might want to configure a custom door, which is the first bit of data, and matches a rule that adds the data that hinges are needed, which leads to a rule that decides what type of hinges.

Within these two basic schemes, there are many application-specific variations one might encounter. A diagnostic reasoning engine might have strategies that let it follow the most likely paths to a solution first, or the least expensive if there are requirements for the user to research more information.

A critical aspect of any reasoning engine is an API that can be called from other application components. This lets logic bases be integrated into an application context in a manner that lets the logic base be updated without requiring updates to the main application code.

Ontology

One of the biggest problems with maintaining a logic base is consistency of definitions. If you are writing a technical support system, for example, and one rule refers to Microsoft® Windows® and another to XP, well they won't communicate unless somehow the system 'understands' that XP is a type of Windows operating system. This, unfortunately, is the difficult part about maintaining a logic base. While it is easy to write and add rules, unless each rule author uses the same terminology the rules will not work as a cohesive unit.

A solution to the naming problem is ontology. Just as with other terms, ontology is a perfectly normal word appropriated for use in computer science. The dictionary definition of ontology has little to do with the computer science use of the word. (Not surprisingly, the term ontology was coined by the same people who decided heuristic was a better word than rule.)

A logic base ontology is a collection of definitions and relationships between entities that can then be used by other components of an application. An ontology would have the information that XP is a type of Windows. And that Windows is a type of operating system. An ontology would also know that Windows 2000, Win2000, and Win2K are all synonyms. Given an ontology, rules can now be entered that refer to XP and be 'understood' to refering to an operating system. For example a rule might have the condition 'if system is an operating system …', and that rule will fire if the value of system is XP. An ontology provides an alternate way to represent logical knowledge relating to terminology that is a powerful adjunct to the more common rules.

Custom Rule Engines

Given the wide variety of ways in which rule engines can represent knowledge, reason with that knowledge, and integrate with the surrounding environment, it is sometimes difficult to choose the right one. A general purpose rule engine will fit a wide range of problems, but might not fit them very well, requiring some stuffing and bending around the corners. For this reason you will find rule engine products designed for specific applications. Microsoft BizTalk® Server is a perfect example. It is a tool designed for integrating business processes. It 'knows' about business process, and passing messages between them, and can be used to express the rules of which process fires when, under what conditions, and which other processes needed to be informed of what when it happens.

There are also products for pricing problems, support problems, configuration problems and a number of other common areas. Each of these will work better for the problem domain they are designed for, but won't be much help for other problem areas.

There is another option to consider as well, and that is the creation of a custom solution for a particular application. Rule engines are not that difficult to write, and building one for a particular application allows for the best possible knowledge representation, reasoning strategy and integration with the main components of the application. The key advantage relates back to semantic gap. A custom knowledge representation language can provide the smallest possible semantic gap for an application with the associated benefits of ease of development and ease of maintenance.

Short Case Studies

In order to better understand the advantages of rules-based solutions, consider the following case studies.

Workflow

A number of companies use logic engines or specialized rule-based tools for encoding logical knowledge about workflow. Typically, these tools integrate with the larger facilities of an application with rules governing workflow.

For example, one large supplier of workflow for the telecommunications industry has integrated the rules describing workflow with the facilities of the larger application context, so the rules can be directly applied to the tasks in the telecommunications domain.

This allows for the separation of the business logic defining work flow rules from the procedural knowledge of the actual processes that need to be performed, and it puts that work flow knowledge in a representation that can be easily understood and maintained.

Configuration

A vendor of windows and doors uses a logic base of rules to drive interactive product configuration through a Visual Basic interface. Contractors use the Visual Basic program to determine the best configuration for a job site, and then automatically connect with the company's server for entering an order.

> "Early AI success stories are related to applications that dynamically gather the data from users, as in diagnostic systems, or that work with organizations that have good computerized records."

They have customized their own development front-end using Excel, allowing the experts to directly maintain the logical knowledge of product configuration using a familiar tool. The spreadsheet is translated to a lower-level rule language that is then used to deploy the knowledge.

Because the logic base is a separate entity from the main application code, it can be easily updated. Whenever the user, working with the Visual Basic program connects to the server, updates to the configuration logic base are automatically downloaded.

The result is a very flexible and responsive architecture for providing their customers, the contractors, with a powerful tool for deciding on and ordering the best products for a particular job.

Mining

A sophisticated pattern-matching application determines if a geologic site has good mining potential. The rules that match geologic characteristics and mining potential are in a logic base that is maintained separately from the Visual Basic interface that graphically displays geologic maps and other information about the potential site. Key to this application is an ontology of definitions of mining terminology that allows geologic field data to be easily accessed by the pattern-matching rules. Without the ontology, it would be very difficult for the rules to make use of the field data entered by different geologists with different ways of expressing the same geologic concepts. The ontology is stored and maintained as part of the logic base.

The application is currently a stand-alone Visual Basic application but will be deployed on the Web using Visual Basic.NET.

Detailed Case Study – Vaccinations

Visual Data LLC provides a Windows software product called *Office Practicum* for paediatrician's offices. It keeps medical records for patients and performs all of the 'data' and 'processing' functions you might expect.

One of the items it tracks for a patient is vaccination history. It turns out that one of the problems for a paediatrician is following all of the complex rules and regulations for vaccinations, and scheduling children for future appointments based on their vaccination needs.

Customers asked Visual Data to provide a feature in Office Practicum that would tell what vaccinations were up-to-date for a child on a visit, and which were due. It should also be able to provide reports on each child analyzing their vaccination histories, making sure they were in compliance with regulations for schools and summer camps. This took Visual Data into the realm of encoding logical knowledge. The knowledge about vaccinations is published in papers made available by the CDC. Each vaccine has one or more schedules of doses, based on the particular type of vaccine, and each has numerous exception rules that describe conditions when a vaccination may or may not be given. There are a number of interesting observations to be made about this application.

Data and Process First, then Logic

The first relates to the Stanford comment about AI in medicine, which was that AI had not advanced due to the lack of data. They observed that AI is really the encoding of logical relationships, but, without entities for the logical knowledge to reason over, there is no practical value in automating the logic. The vaccination program illustrates this.

People in the past have worked on AI systems to automate vaccination logic, but the patient data on vaccination history was not readily available. It had to be typed in by hand as input to the system in order to get a vaccination schedule. However, any medical practitioner experienced in vaccinations could figure out the schedule directly from the data in about the same time without having to engage a computer in the process. So there wasn't much point.

Office Practicum provides enough help in the day-to-day business of running a paediatrician's office that collecting data on patient histories comes naturally. Because that data is in the computer, and because the office is already using the computer for other aspects of managing the patient, it now makes sense to automate the logical knowledge for vaccination scheduling. In fact, it was the customers who started to ask for this feature, after using the software. They noted that all the vaccination information was in the computer, so why couldn't it automatically generate the vaccination schedules.

Procedural Code Works, but is Impractical

Visual Data first attacked the problem by attempting to encode the vaccination logic using procedural code. In their case the application is developed in Borland's Delphi, and they used Pascal for the encoding. The software worked, but was difficult to write, and was in a large complex module, and only provided some of the features they wanted to provide.

However, the world of vaccines kept changing. New vaccines were coming out that combined earlier vaccines in a single vaccination with new more complex rules about the interactions between the components. Customers wanted to know when the software would support Pediatrix, a new complex multi disease vaccine. The software developers groaned.

While they were a Delphi shop, and familiar with Delphi, and would love to do all their work in Delphi, they realized the vaccination module was just too difficult to maintain, so they opted for a logic base solution. The logic base reduced the code size from thousands of lines of code to hundreds of lines of easily understandable rules. It was the same 10:1 ratio seen so many times for these applications.

Further, the rules were now in a format that their resident paediatrician, not a programmer, could understand. The application was restructured so that the Delphi code called the logic base, much the same way it called the database. The 'knowledge' of vaccination scheduling was now completely outside of the core Delphi code. The logic base can be updated without affecting the main application, just as the database can be updated without changing the application.

Unlike the database, the logic base must be tested, and Office Practicum uses a tool set to independently test the rules. Regression tests are a part of the system, so that various scenarios can be automatically retested when changes are made to the logic base.

The Nature of Vaccine Logical Knowledge

Visual Data did not use an off-the-shelf rule engine for a couple of reasons. One was cost, but more important, the logical knowledge of vaccines seemed to require its own specific set of ways to represent knowledge. These included definitions, tables and rules. While all three could be stored in rules, some of the visual clarity of the mapping from documentation to logic base would be lost. It would be better if the logic base more directly expressed the CDC logic.

Further, most of the logical knowledge had to do with dates and date intervals expressed in any of the date units – days, months, weeks or years. The conditions in the rules needed to use the intervals and dates correctly, and assume age from context when appropriate.

Accordingly the knowledge representation language for the vaccination system was designed to have

– An ontology of terms to store definitions.
– A means of entering tabular knowledge.
– A means of entering rules.
– Language statements that recognized dates and date intervals.

This made it easy to add new definitions without affecting the rules, allowed for the direct encoding of the basic tables, and enabled the rules to refer to the tables and reason with concepts such as the last live virus vaccination.

Ontology

The ontology describes semantic relationships such as the various types of vaccines that are Hib vaccines, as well as distinguishing between those that contain PRP-OMP and those that don't. This is critical because different schedules are used for each.

It also describes which vaccines are live virus vaccines, another critical fact used in many of the rules concerned with the interaction between different vaccines that both contain live viruses.

Additionally, there are multi-vaccine products, such as a combined measles, mumps and rubella (MMR) vaccine. There are rules that are just concerned with, for example, whether or not a child had a measles vaccine, but the database might indicate MMR. This knowledge is all in the ontology as well.

Here, for example, are the definitions in the logic base that indicate Varicella and Small Pox are live virus vaccinations:

```
live_virus ->> 'Varicella'.
live_virus ->> 'Small Pox'.
```

Here are some different types of Hib.

```
'Hib' ->> 'HbOC'.
'Hib' ->> 'PRP-OMP'.
'Hib' ->> 'PRP-T'.
```

Tables

Standard tables provide the minimum

> "The ability for an organization to successfully encode its logical knowledge can lead to better services for its users."

age, recommended age, and minimum spacing interval for each dose of a vaccine. If this was all there was to the vaccination logic, then a database solution or other table lookup would have worked, although even the tables aren't that simple. For a given vaccine, different tables apply depending on factors such as whether it is a multi-vaccine, what the active components are, and whether or not the child has followed a standard schedule.

Here's an example of a table in the logic base that describes the Hib schedule for vaccines containing PRP-OMP.

```
table('Recommended B', [
% Recommended Schedule B from
'DHS Hib 2003Mar' for vaccines
% containing PRP-OMP
% Dose Minimum    Minimum  Recommended
%      Age        Spacing  Age
  [1,  6 weeks,   none,    2 months],
  [2,  10 weeks,  4 weeks, 4 months],
  [3,  12 months, 8 weeks, 15 months]]).
```

Rules

The rules work in concert with the definitions and the tables. They are used to determine which table is appropriate in a given situation. They also provide coverage for all the exception cases, such as the fact that a given vaccine isn't necessary after a certain age, or that a schedule can't be kept if other live virus vaccines have been given, or what the corrective measures are if a previous vaccine was given earlier than allowed.

Here's a relatively simple rule that fires when the Polio sequence is complete. Note that the ontology lets rules refer to either 'Polio' in general, or the two main vaccines, 'IPV' and 'OPV' separately. This rule describes when an OPV sequence is complete. The output includes an explanatory note that is displayed in verbose mode.

```
complete :-
    valid_count('OPV') eq 3,
    vaccination(last, 'OPV')
*>= 4 years,
    !,
    output('Polio', [
    status = complete,
    citation = 'DHS IPV
2003Mar',
    note = [
        `Complete: An all OPV,
         three dose sequence is
         complete when`,
        `the last dose is given
         after 4 years of age.`
         ]]).
```

Modularization

Modularization was a key requirement for this application. The tables and rules for each vaccine were kept in separate modules. The ontology, on the other hand, was in a common module as it was used by all the other modules.

Reasoning Engine

The reasoning engine for the vaccine logic base is designed to meet a variety of application needs. It takes as input the vaccination history of a child and then goes to the module for each vaccine in question and gets the status information for that vaccine. This includes an analysis of the past vaccinations with that vaccine; the status as of today, the current office visit; and the recommended range and minimum dates for the next vaccination with that vaccine.

Each module is designed with the same goal and output, and that goal is called by the reasoning engine. This allows for the easy addition of different vaccines, and the easy maintenance of any particular vaccine.

The reasoning engine has an application program interface (API) that is used by the calling application. The API provides the various reports required for different uses. For example, it can tell what vaccines need to be given on the day of an office visit, or what vaccines will be needed for scheduling a follow-up visit. It also allows for short and verbose reporting and explanations of the recommendations, and provides the historical analysis reporting required for camp and school forms.

Cost Benefit

The benefits from the logic base approach have been:

- a 90% reduction in code used for vaccine logic rules,
- direct access to the knowledge by the in-house paediatrician,
- localization of all the vaccine logic, which used to be scattered in the different parts of the application with different needs,
- easy maintenance, and quality assurance testing, and
- additional capabilities that were too hard to encode before, such as the complete analysis of past vaccination history and support for new multi-vaccine products.

All of these benefits add up to the one major benefit, which is that their software now provides better services for their customers in this area which is critically important in the running of a paediatric office.

The costs were:

– time spent investigating and learning about various alternative approaches for encoding the vaccination logic,
– software license fees,
– two month's development time, and
– time spent learning the new technology.

Conclusion

Logical knowledge, unlike factual or procedural knowledge, is difficult to encode in a computer. Yet, the ability for an organization to successfully encode its logical knowledge can lead to better services for its users. The question then, is how best to encode logical knowledge. It can be shoe-horned into data–and procedure-based tools, but the encoding is difficult, the knowledge becomes opaque, and maintenance becomes a nightmare.

Rule-based and logic-based tools are better suited to the encoding of logical knowledge, but require the selection of the proper tool for the knowledge to be encoded, and the learning of how to use that tool.

Off-the-shelf rule or logic tools sometimes provide a good solution, but often the knowledge representation of the tool doesn't fit the actual knowledge, or the reasoning engine doesn't use the knowledge as it is supposed to be used. This leads to the same coding and maintenance problems experienced with conventional tools, but to a lesser extent depending on how big the semantic gap is between the knowledge and the tool.

A viable alternative is the building of a custom logic-based language and reasoning engine. This allows for the closest fit between the coding of the knowledge and the actual knowledge, and for the cleanest integration between the tool and the rest of the application context.

Resources

http://ksl-web.stanford.edu
Standford's Knowledge System Laboratory's home page has information about their current work with ontologies and other research areas, as well as links to related sites and organizations. *www.aaai.org*

American Association of Artificial Intelligence (AAAI) is a non-profit scientific organization devoted to supporting AI research and development. They have conferences and journals and an excellent Web site, *www.aaai.org/AITopics/aitopics.html,* that introduces AI research and topics.

http://www.ddj.com/topics/ai
Dr Dobb's Journal's Web site devoted to AI topics and links.

www.ainewsletter.com
Past issues of the DDJ AI Expert newsletter. You can subscribe to the newsletter at *www.ddj.com*

http://www.pcai.com
PCAI Magazine has a wealth of information about AI technologies and products on their Web site.

A Google search for 'business rule engine' will yield links to a number of commercial offerings.

Dennis Merritt
dennis@amzi.com

Dennis Merritt is a principal in Amzi! Inc. (www.amzi.com), a small privately-funded company specializing in tools for developing knowledge-based and AI components that can be embedded in larger application contexts. He has written two books and several articles on logic programming and expert systems, and is currently the editor of the Dr. Dobb's Journal AI Expert newsletter www.ddj.com

DasBlog: Notes from Building a Distributed .NET Collaboration System

By Clemens Vasters, newtelligence AG

Keeping a diary seems mostly an obsession of teenage girls, politicians, and people who are planning to write their memoirs at a later point in life. A diary is the place to write down a daily or weekly snapshot of very private thoughts that are nobody else's business. No matter whether the diary is kept by a 15 year old girl who is desperately in love with rock star Robbie Williams, or by the German chancellor, all diaries have one thing in common – they are top secret, locked away and never shared with anyone.

Looking at the newest trend on the Web, we might need to revise, or at least widen, our definition of a diary. Weblogs (usually shortened to 'Blogs') are the digital equivalent of the personal diary, and have taken the Internet by storm over the past two years. Mid-2003 estimates[1] put the total number of Weblogs somewhere between 1.3 and 2.2 million, and these numbers are rapidly growing. The two most striking differences between diaries and Weblogs are that Weblogs are not only for teenage girls or politicians, and that they are not secret at all. On the contrary; Weblogs are out there for everyone to see!

So why is this topic being discussed in a magazine for software architects and information technology managers? There are two main reasons: First, there are a lot of architectural lessons that can be learned from the Weblog phenomenon and from the technologies that make the Weblog universe tick. In fact, the Weblog space as a whole has already grown to be the largest distributed XML and Web services application in existence. Second, Weblogs are becoming a strategic tool to improve communication and collaboration in the enterprise that may eventually turn out to be just as important as email.

In the first part of this article, I will analyze and explain the Weblog phenomenon and give some examples of how Weblogs can be used as collaboration systems in business environments. In the second part I will take a closer look at Weblog related technologies, examine a concrete example of a Weblog Engine in the form of newtelligence's free Microsoft ® ASP.NET based dasBlog application, and share some of the architectural lessons newtelligence learned from implementing, deploying and running Weblogs on this platform.

Part 1: The Weblog Phenomenon

Before we go into the more technical aspects, we should spend a bit of time analysing why Weblogs have become so wildly popular and why some people are publishing their once secret diary on the Internet, where it can potentially be read by anyone.

Keeping a diary, and therefore keeping a Weblog, is a very good idea, indeed. Formulating thoughts and ideas in writing requires more serious and intense consideration of a topic than just thinking about it. Keeping track of thoughts, ideas, problems, and solutions over time helps build a great growing resource of personal experience that you can turn back to in order to look up all of those details that were once well thought out and known but now almost forgotten.

However, being a good idea is not enough of a reason to explain the sudden popularity of Weblogs. What makes Weblogs really popular is that they provide a near-perfect personal publishing approach for everybody. Even with the most recent generation of HTML editing tools, maintaining a private Web site is far too complicated for most people and the result is often disappointing when compared to the polished Web sites created by professional web-designers. Even if the technical design challenges can be overcome there is often a 'content dilemma' regarding the information that should actually be included in the home page. Unless the owner is a fanatic hobbyist keen to share his expertise about postage stamps, model railways or his favourite movie star, the home page creation efforts usually result in little more than a few pictures, a personal greeting such as 'Welcome to my home on the Web' and some favourite links – with the frustrating consequence that nobody will ever look at it.

Weblogs – or more precisely, the tools that are used to create Weblogs – help overcome these hurdles to personal publishing. Most popular Weblog tools are in fact quite sophisticated personal content management systems that will take care of rendering content into a professional looking HTML template. They usually provide content categorization as well as topical and historical navigation capabilities. This functionality, together with their ease of use also seems to solve the content dilemma. If the effort to publish a quick thought of no more than three lines onto a Web site isn't greater than that of writing an email, then it's more likely that it will happen. The ease of publishing and the ability to publish random thoughts quickly, results in a qualitative difference between home pages and

"The Weblog space as a whole has already grown to be the largest distributed XML and Web services application in existence."

[1] http://www.blogcensus.net/ and http://dijest.com/bc/2003_06_23_bc.html

Weblogs: Home pages help show off HTML skills or knowledge and admiration for a particular subject; Weblogs exhibit personality and tell an ongoing story.

Moreover, Weblogs have turned into a public discussion platform where thoughts and ideas can be exchanged. Because Weblogs are formatted as hypertext, they allow links to other Weblogs, enabling Weblog authors to publicly comment on other authors' entries, and to any other Web-pages the authors want to highlight to their readers. Many Weblog tools also facilitate the collection of per-topic reader comments, allowing everybody, not only 'Bloggers', to participate in discussions. Weblog tools not only expose the content as Web sites, but also in XML. The most commonly used format is the Real Simple Syndication (RSS) 2.0 format (also referred to as RDF Site Summary or Rich Site Summary).

XML and RSS allow Weblog readers using special tools to consolidate all content of interest into a local content store that's searchable, easily navigable, and provides pre-categorized views that are also available offline. The NewsGator[2] plug-in for Microsoft Outlook® even goes so far as to integrate this ability into your everyday email client.

Weblogs in the Software Business
Recently, Weblogs have moved very much beyond being just a trendy tool for personal publishing and community discussion. Corporate knowledge and information workers already benefit greatly from Weblogs as an information source today. Software developers and architects as well as sales and marketing employees quite often find Weblog links amongst the top 10 entries of a Web search result – and sometimes Weblog entries seem to be the only good source for certain information.

Technology companies like Microsoft and Sun even have dedicated portals where their employees can host their Weblogs. However, while the Microsoft portal is quite liberal about Weblogs and only provides a set of links pointing to their employee's personal blogs reflecting their personal opinions and not the 'official' Redmond position, the Sun portal and Weblogs are much more of a developer marketing and developer education tool with a personal touch. Both approaches have upsides and downsides and neither can be called better or worse. Sun's primary interest is a consistent, polished external corporate message, while Microsoft's approach is to get and retain traction with customers on an informal, personal level and give them an uncensored insight about what's happening behind the scenes.

Both the Microsoft and Sun approaches to public corporate Weblogs fill a gap in corporate messaging. The Weblog format, the personal nature, and the ease of publishing allows authors to post small, informative snippets about a product or solution path and, as demonstrated by the very liberal Microsoft approach, sometimes even without going through the usual corporate review, editing and publishing cycle. It's obvious that even companies as large as Microsoft cannot document every problem solution and workaround using the 'official channels', with all of the requirements around localization, consistency and publication processes – letting employees augment the official documentation with their own insight is a brilliant way to complete the picture and fill those gaps.

The corporate use of Weblogs is most visible in the software industry, but it's not restricted to it. There are signs that Weblogs are starting to fulfil similar roles in other fields of business such as law, the media industry, the fashion industry and various fields of engineering. Weblogs aid in bringing a more personal touch to the corporate message and are a very valuable marketing and public relations tool that demonstrates the personal competence and abilities of the people that are the drivers behind the a company.

Weblogs as an Enterprise Communication and Collaboration Tool
An even more attractive application of Weblogs for corporations is information distribution and collaboration inside the corporate network. Although many enterprises have embraced and implemented portal solutions and 'groupware' as information distribution and discussion platforms for a long time now, Weblogs have the potential to substantially enhance the corporate information ecosystem and foster a more transparent corporate culture.

Let's look at a few examples of how Weblogs and related technologies can be used in a corporate environment and that use differs from more established solutions:

– **Personal Weblogs** Personal Weblogs are the most obvious

[2] NewsGator: http://www.newsgator.com

application, but – depending on the corporate culture – also have the most potential for conflict. Personal Weblogs are a 'personal portal' and allow individuals to track the results of their own work and information they gather from third party information sources such as external websites. Due to its chronological nature, it also provides a great way of document the history of decision processes, of assumptions about future events and therefore provides a great foundation for post-mortem analysis in the event of success or, more importantly, failure. The conflict potential lies in issues like content ownership, supervision, freedom of personal expression, disciplinary consequences and similar issues.
- **Topic-Centric Knowledge Weblogs (K-Logs)** Knowledge Logs are not person-centric but topic-centric. They are based on the same technology as Weblogs, but have multiple authors, usually from one team, but sometimes across teams or even divisions. K-Logs focus on certain subject areas and enable the aggregation of information and references to topical content as a growing and chronological repository. In this function K-Logs have substantial overlap with classic knowledge portal solutions. What makes K-Logs appealing is the relatively low software acquisition cost, the ease of use and the ability to distribute and aggregate the content via RSS. These advantages will be examined in detail later in this article.
- **Team Weblogs** Team Weblogs are team centric and track the progress of the development of a certain product or project.
- **Automated Weblogs** Using features like 'Mail-To-Weblog' (discussed later in this article) or integrating with Web services exposed by Weblog engine software, Weblogs can serve as an easy-to-install and easy-to-maintain publishing point for automatically generated information. In the software industry this information could include daily reports generated from automated build and test processes, in manufacturing processes it could be statistical information, and so on. The benefit of automated Weblogs is that the effort for publishing such information in an accessible way is minimized and it is sufficient for the information provider to supply very simple plain-text fragments.

An analysis of your own corporate knowledge capturing and distribution needs might yield more possible applications of Weblogs. In addition to this, subscription-based RSS information services can help to improve information distribution inside the company without flooding email inboxes. Examples of this are automatically generated daily or hourly reports about system or machine activities, but also mundane yet important things such as today's menu in the company's cantina or the latest scores of the company football team.

Part 2 Weblog Technologies and Applications

The 'Blogosphere', as the Weblog space is also often called by Weblog aficionados, is powered by a set of core technologies and techniques that we need to explain before we can move on to the details of the concrete implementation in newtelligence's dasBlog.

RSS Publishing and Aggregating Information via XML

The most important Weblog technology is undoubtedly the Real Simple Syndication[3] (RSS) XML format that has already been mentioned earlier in this article. RSS was initially created by Userland Software and Netscape as the XML format behind the 'Sidebar' feature of Netscape Navigator 6.0 that was the follow-up technology to the 'Netcaster' in the 4.0 generation of Netscape's product. RSS can be seen as a response to Microsoft's XML-based Channel Definition Format[4] (CDF) for 'Active Channels' and the 'Active Desktop' that had already been introduced in Internet Explorer's version 4.0, and both serve approximately the same purpose:

The common idea behind the RSS and CDF was to provide machine readable indices for websites that could be picked up by Netscape Navigator and Internet Explorer and allowed the browsers to display the current site highlights and headlines either in the Netscape Sidebar, in Internet Explorer's Favourites View or on Windows' 'Active Desktop'. The vision and promise was that all news headlines and articles that a user was interested in could be imported in a quick online session and were then available offline – primarily as a convenient workaround to expensive, pay-per-minute Internet access cost. The long defunct Pointcast Network[5] had a similar approach and was also (in part) using the CDF format to acquire information from its sources and to push subscribed channels to their client software. Unfortunately, none of these products and features were blessed with any great success; Pointcast went floating belly up and

[3] RSS 2.0 specification home: http://blogs.law.harvard.edu/tech/rss

[4] CDF W3C Note: http://www.w3.org/TR/NOTE-CDFsubmit.html, Official reference: http://msdn.microsoft.com/workshop/delivery/cdf/reference/CDF.asp

[5] Pointcast Webopedia entry: http://www.webopedia.com/TERM/P/PointCast.html

'channels' built for either Netscape or Internet Explorer have become an extinct species. RSS survived thanks to the continued efforts of Userland Software and some other vendors focusing on small scale content management systems, from which today's Weblog tools eventually evolved.

Despite its popularity and the fact that RSS has accumulated a critical mass of adoption that makes it hard to replace, it is widely recognized that RSS has several critical deficiencies requiring changes or additions. RSS lacks proper support for XML Namespaces, does not use the ISO time format mandated by the XML specification, has no normative XML Schema or even Document Type Definition, and the specification itself is ambiguous and lacks formality. These issues have prompted the formation of a working group[6] around the IBM engineer Sam Ruby who are working to replace RSS, along with a consolidation of most of Weblog technologies into a set of specifications under the name *Atom*.

Referrals Sparking Discussion and Interaction

The public interaction between Bloggers that emerges in the Blogosphere is one of its greatest appeals and motivators. Discussions on certain topics quite often involve dozens of authors who independently publish their own views on their Weblogs, but by citing other authors and linking to their respective Weblogs they jointly create a hyperlinked mesh of views, information and opinions on a given topic.

Discussions spanning multiple Weblogs are formed by no rules other than chaos. This is distinctively different from discussions in Internet newsgroups, on mailing-lists, or in public folders in a Groupware system. In these, discussion participants must active subscribers of a certain group and the discussion usually goes unnoticed outside of the group. Google's newsgroup archive exposes newsgroup discussions to the web to some degree, but still requires the user to explicitly search either in a particular newsgroup or for certain keywords.

The primary tool aiding the chaotic formation of discussion and interaction is a simple and well-known mechanism supported by all common Web browsers: The HTTP *Referrer* header. When someone comes across an interesting tip, a thought provoking article or an opinion they post their applause, concerns or supporting information to the own Weblog – and in doing so add a hyperlink to the cited Weblog post over on the other Weblog. Because almost all Weblog tools recognize and log the *Referrer* HTTP header, notifying the author of the original entry is as simple as clicking the link in a browser, because the *Referrer* header contains the URL of the page where the hyperlink was set. Most popular Weblog engines track and consolidate the referrals in easily accessible lists, ranking them by the number of visitors that have arrived at the Weblog through the external links and rendering the referrer URL as a clickable hyperlink – some Weblog engines can even notify their owner by email of every such referral. In that way, an author learns about external comments or citations and can post his own responses or additional comments – and in the process linking back and possibly citing and linking Weblogs of third parties for examples or supporting opinions, causing the on-topic mesh to form and spread.

Because hyperlink referrals still require manual intervention in order to trigger notification, two additional and more instant notification mechanisms have gained the support of the Weblog community and tool builders: Pingback[7] and Trackback[8].

Pingback allows implementing automatic notifications between Weblog engines without having to rely on HTTP referrals. Pingback defines a Web service interface (using the XML-RPC Web services protocol, not its successor SOAP) and two auto-discovery mechanisms. The function principle is very simple: When the Weblog author posts a new entry to their Weblog, the engine looks at the submitted HTML fragment and scans it for hyperlinks. It will then issue an HTTP GET request to each of those links, using one or both of the auto-discovery mechanisms, looking for an HTTP header or a special tag embedded in HTML, in order to find out whether the link target supports the Pingback protocol. If a Pingback endpoint is detected, the engine will submit a ping Web service call, supplying the URLs of both, the pinged and the pinging Weblog entry. Pingback has the advantage of instant notification about citations and, just as important, about changes to these citations.

Trackback aims to provide similar functionality, but with a slightly different spin. The protocol does not only provide the URL of the pinging entry, but optionally also the title

[6] Project Atom: http://www.intertwingly.net/wiki/pie/FrontPage

[7] Pingback 1.0: http://www.hixie.ch/specs/pingback/pingback

[8] Trackback 1.1: http://www.movabletype.org/docs/mttrackback.html

and a short excerpt of the source entry along with the Weblog's name. Contrary to Pingback, which is fully automatic, Trackback is typically used as an explicit, on-demand notification mechanism.

The major technical difference between Pingback and Trackback is that Pingback employs an XML-RPC Web service interface while a Trackback ping is technically equivalent to submitting a form in a browser – the information is posted using a HTTP POST request employing the *application/x-www-form-urlencoded* content type, precisely as it is the case with HTML forms. Although different, both protocols succeed in achieving their goal: improving collaboration and communication.

To enable people who do not own their own Weblog to participate in Weblog discussions,, most Weblog engines support user comments that can be added using a Web-based interface. Additionally, a widely adopted Web services interface for comments exists; the Comment API[9]. The Comment API is directly supported by some of the popular RSS aggregators like 'RSS Bandit'[10], which allow readers to post comments straight from the tool.

dasBlog: Implementing a Weblog engine

Early in 2003, at newtelligence we decided that building our own Weblog engine would be a good thing to do. There were several motivations. As the most active and likely best known Weblog author at newtelligence, I primarily wanted to have a replacement for my previous Weblog tool for myself, with the side effect that the other colleagues at newtelligence could use it too. Writing our own blog engine also promised to give us a great set of example code to use for developer education and a platform to try out new technologies and techniques. Finally, developing a solution that supports all the described and a few more collaboration Web services seemed like a great experiment to participate in, and allowed us to research the reality of a distributed system that already implements a great deal of the Web services vision.

At the time I got around starting to implement dasBlog in July 2003, with just 5 calendar weeks allocated to complete the job, there were two major Weblog engines existing for the Microsoft .NET Framework, our default platform: The engine powering http://weblogs.asp.net (now called '.Text'), for which code was not available at the time, and the engine BlogX, which had been thrown together by a couple of people at Microsoft in their spare time along with some community contributors.

The 'make or take' decision was a relatively easy one, because BlogX was already a working implementation with a file-based backend store and was relatively lightweight in terms of existing features providing a good skeleton that made refactoring and adding new features relatively easy. Also, the license conditions for BlogX were largely equivalent to the BSD license, which we also favour for work that we publish for free and in source code form.

While our initial intent was to merge our changes back into the original BlogX code base, it turned out that the refactoring process led to the elimination of almost all of the original BlogX source code as the project progressed. Because merging the result into the code base would have amounted to a hostile takeover of that community project, we've decided to give it a new name and to maintain it as a separate project, and so dasBlog was born. Version 1.0 of the new code base went public after 3 weeks, with the follow-up versions leading up to the complete feature set in version 1.2 being released after 5 weeks – in time and with a stability and quality that has convinced several dozen bloggers to abandon their old tools and switch to the dasBlog engine even in the early stages of the project.

dasBlog: Requirements, Considerations, and Solutions

Fundamentally, dasBlog is a small content management system that's directly bound to a rendering engine, which renders all content just in time and based on the view that a visitor chooses.

Storage

The primary task of a Weblog system is to capture and present a chronology of events. The front page of a Weblog therefore presents a configurable number of Weblog entries, chronologically ordered, with the most recent entries at the top. This fundamental principle also requires that visitors can easily navigate through the history of the Weblog by date. This primary function immediately influences the design of the backend store, for which the most common lookup criteria is a date or a time span. At the same time, it must be possible to efficiently access individual Weblog entries by their identifier in order to

[9] Comment API: http://wellformedweb.org/story/9

[10] RSS Bandit: http://www31.brinkster.com/rssbandit/

attach 'tracking' information such as referrals, pingbacks and trackbacks, and to associate and display comments as explained earlier in this article.

Because a Weblog is a person-centric, not topic-centric publishing point, it is also required to enable and ease by-topic navigation by introducing a categorization scheme. Creating and maintaining categories should be largely automated and should not require much administrative effort on behalf of the user.

Fulfilling these requirements would be very easy with a relational database system and a few simple indexes. However, to achieve the desired ease of initial deployment and future upgrades coupled with minimal administrative effort for anyone with a low-cost, ASP.NET enabled account at a web-hosting company as well as for users on corporate desktop machines running a local web server as their own publishing point, it's not a good idea to depend on the existence of a full database system. Instead, the backend is factored in a way that a database could be supported if that requirement should arise, but the best and primary storage mechanism is quite simply the file system. This could have been achieved by using a file-based database like Microsoft's well-known Jet or FoxPro engines, but such a built-in dependency would limit extensibility and impact the efforts required for upgrading to newer versions of the software as they appear. Once the road down the database route is taken, any changes or additions to the storage require schema updates for databases in the installed base, substantially increasing the administrative effort.

The resulting architectural decision that was already pre-defined by the original BlogX code base, and which we consequently decided to stick with, was to store all information in XML files in a subdirectory of the application. Because the lookup criteria is based on time, or at least a time interval, the content is stored following a 'one day, one file' scheme and the index is simply the file system's directory information: The files names contain the date. The auxiliary indexes for the categories and the entry's unique identifiers are stored in separate files. Additional information such as tracking data (referrals, pingbacks, and trackbacks) and comments are stored in files that are also named (and thus indexed) by date, but are kept separate from the actual content in order to limit concurrency issues and to address the differences in their characteristics:

The core content has a very low update frequency (a few times a day), has very many reads, and must never be lost. Tracking information is updated very often, potentially concurrently, has many reads, and is less critical. Whenever changes to the core content occur, the engine persists them synchronously to be able to report any errors straight back to the user. These changes also cause all in-memory caches to be discarded and the auxiliary indexes to be rebuilt. All trackings, however, are processed asynchronously on a single secondary thread that is sequentially fed information through an in-memory queue. This can be done, because the timeliness requirements for trackings are very relaxed: They need to be reflected in the Weblog eventually, but they don't need to appear as the event occurs.

Because a new file is created each day, the resulting file sizes are quite small (typically substantially less than 100KB), and writes are quick with minimal locking. An aggressive approach to caching allows synchronous updates to the in-memory caches and the backend store especially for the tracking information, and therefore further reduces concurrency problems. It should be noted, however, that the chosen storage model and the interaction between the in-memory caches and the backend store limits clustering or otherwise having multiple engines share a common store. That's a deliberate and acceptable restriction, because even very popular Weblogs usually get only a few ten thousand hits per day. This is aided by the fact that most users read Weblogs though RSS aggregators and due to infrequent updates of the core content, the RSS streams can be easily cached on the server side and even by upstream proxies.

Content Management

While we have already discussed the storage strategy, we haven't yet covered how content is actually submitted to the engine. Here again, dasBlog had to fulfil multiple requirements for a variety of different usage scenarios.

The most obvious way to submit content into a Web-based application is to use a form on a Web page. dasBlog supports this for all browsers, but gives Microsoft's Internet Explorer preferred treatment for a few quite simple reasons: Users that access the editor web pages using Microsoft Internet Explorer are provided with a page that includes a set of client scripts and an inline frame, utilizing Internet

"All trackings, however, are processed asynchronously on a single secondary thread that is sequentially fed information through an in-memory queue."

Explorer's inherent ability to act as an HTML editor. With this, users get rich, in-browser text editing capabilities and can style text using several fonts, typographic effects and colours. The editor also supports attaching files and embedding pictures. The binaries are uploaded using the standard HTML upload control and stored in a special directory below or alongside the content directory. Once the upload is complete, the picture or a hyperlink is inserted into the current text.

For users with other browsers, such as Opera or the Mozilla browser, the web-based editing capabilities are unfortunately much more restricted. With these browsers, users only get a standard multi-line text field and must write HTML mark up explicitly. The decision to go with such a limited version for non-Microsoft browsers is based on Internet Explorer's market share, the assumption that the users of dasBlog will run Windows on their desktops, and the fact that HTML editing support is not standardized across browsers. However, this limitation isn't as significant as it might appear at first sight, because using the web form is only one of three ways to submit content into the engine. The first alternative to submitting content through a browser is to use one of a variety of offline Weblog editors such as Zempt[11] or w.bloggar[12] that directly support the Web services API developed by the makers of the popular Weblog environment 'Blogger'. Any editor that can target Blogger servers can also target dasBlog. dasBlog implements the Blogger API along with extensions made for the competing MovableType Weblog software and extensions made by Userland, so that a wide range of tools can be used both to submit new entries and to edit existing entries from a rich client.

The Blogger API and its various extensions define Web services endpoints that do not use the SOAP protocol, but rather use XML-RPC. The Atom working group that has already been mentioned in the previous discussion on RSS is planning to consolidate the partially overlapping functionality of what are essentially three APIs and define the resulting API to be SOAP based. However, the server-side XML-RPC protocol endpoints are sufficiently easy to target by client applications, because there are plenty of pre-built libraries supporting the protocol for practically all platforms that matter[13], even if these libraries are not too well known. This entry point to the content management backend is not only well suited for interactive editing tools, but also as an interface to push automatically generated content into dasBlog from any other application and even to synchronise content between Weblogs.

The second alternative to submitting content through a browser, and by far the most attractive one, is email. When the dasBlog web application starts, it spins up a dedicated thread that will watch, in configurable intervals, a POP3 mail account defined by the Weblog owner. Whenever the engine polls the account, it processes every email item in the account, looking for emails whose subject line is prefixed with a configured passphrase. Emails with a matching passphrase are added to the content store. This is obviously a minimalist security measure and the passphrase can even be empty, allowing any email to be published. dasBlog can handle HTML and plain-text formatted messages, and extract, store and link attachments, handle embedded pictures, and through a configuration switch create and embed thumbnails for picture attachments.

Email support provides the most flexible and instantly interoperable model for adding information to the Weblog, is readily supported on every platform. It allows content creation and submission anywhere, with familiar tools including SMS and MMS messages sent from mobile phones through an email gateway. With support for HTML and picture embedding and using state-of-the-art email tools like Microsoft Office Outlook 2003, publishing rich content to the Web becomes easier than ever.

Rendering Engine and Localisation

The rendering engine is responsible for formatting the content for Web presentation. The core requirements are easy to define: Navigation through the site should be easy and obvious for all visitors, the site should be accessible using virtually every current Web browser on any platform, and it should be possible to easily customise and enhance the site's visual design.

These requirements led to an approach where dasBlog borrows heavily from the popular Radio Userland[14] Weblog-tool, and is indeed largely compatible with design templates created for that tool. The reason for taking this route was that there are many free and ready-to-use design templates available for Radio Userland, and also for Userland's Manila[15] content

[11] Zempt: http://www.zempt.com

[12] W.bloggar: http://www.wbloggar.com

[13] XML-RPC implementations: http://www.xmlrpc.com/directory/1568/implementations

[14] Radio Userland: http://radio.userland.com

[15] Manila: http://manila.userland.com/

management system, and allowing reuse of these immediately provided a variety of widely known and appealing visual themes that fit many personal tastes. In addition to having a broad selection of ready-to-use themes, the basic navigation scheme is therefore also aligned with a large number of other public Weblogs, providing the desired instant familiarity and ease-of-use.

The installable version of dasBlog comes with a set of these templates already configured for use – enabling the user to focus on content and not on technical details of HTML right from the start. Still, because many users want to give their Weblogs a personal touch and are not afraid of HTML, customisation beyond simply selecting a template must be very simple.

Design templates for dasBlog use a combination of simple HTML, Cascading Style Sheets (CSS) and a set of macros that is implemented by the engine. A design template (or theme) always consists of three simple HTML files: homeTemplate.blogtemplate (or .txt or .html), dayTemplate.blogtemplate and itemTemplate.blogtemplate. The homeTemplate is used to render the content framework for every page, the dayTemplate is used as a frame for the content of a certain day and the itemTemplate is used to render individual entries. The engine also supports separate templates for each content category.

The templates themselves are fairly easy to customize using a standard HTML editor with Cascading Style Sheet support. The supported macros are well documented[16] and can be inserted into the page using special escape sequences. All dynamic elements such as the calendar or the category lists are partially hard-wired into the rendering engine because of their complexity, but their appearance can be extensively customized using Cascading Style Sheets.

Another concern and requirement for dasBlog was to have good support for localization. Because newtelligence is a German company with customers all across the EMEA region, it was important for us to support full localization into German and English for ourselves and into all EMEA region languages, including the right-to-left languages Hebrew and Arabic, for our customers. To make localization work, dasBlog combines several techniques.

The engine looks at the *Accept-Language* HTTP header that all major browsers send with each request to indicate the user's preferred language. The current culture of the ASP.NET thread handling the request is set to the language-identifier with the highest preference and subsequently causes all resources to be loaded from the most appropriate resource tables, with a default fallback to a neutral culture that contains all resources in English. This causes the date formatting and the calendar to be properly localized from within the .NET Framework itself and causes all hardwired strings that the Weblog engine needs to emit to be rendered in the most appropriate language. For templates, the engine provides a macro that allows specifying multi-lingual strings within the HTML source. Switching the thread into the appropriate locale also enables right-to-left support for Arabic and Hebrew, because the resource tables for these locales contains a special flag that causes the engine to inject the appropriate additional *dir* attributes into the HTML streams.

In addition to these general localization techniques, it is possible to explicitly set the language for every Weblog entry so that only visitors who have this language listed in their browser preferences (and therefore indicate that they can understand it) will see this content. If an entry uses the invariant culture default setting, it is shown to all visitors, independent of language preference. If a language is set of individual entries this is also reflected in the XML data streams rendered by dasBlog where the respective elements are labelled with the appropriate *xml:lang* attribute.

The combination of these techniques demonstrates that flexible localization is very possible for Web sites in general and we found that the effort it took to implement the complete localization support was very low when using the .NET Framework. In fact, the code required to make localization work was added to the already existing application and deployed in less than two days.

Threading Model and Hosting
dasBlog is hosted in an application domain inside the ASP.NET worker process on Windows 2000 and Windows XP, or in the Web Application runtime of Internet Information Server 6.0 on Windows Server 2003. Because of the focus on a single user's Weblog and the resulting limited traffic, it is safe to minimize I/O workload by using aggressive caching, incrementally loading content at the time of the first request and keeping it cached for

[16] DasBlog Macro Documentation: http://www.dasblog.net/documentation/CategoryView.aspx?category=Templates%20and%20Macros

further requests. Because users very rarely browse through the Weblog's history, typically only the content of the last month is cached and even for a very busy Weblog this means an in-memory data volume of well under 2MB. Requests for binaries and pictures are handled and served by the Web server directly and are therefore not a concern for the engine.

Because of these considerations, operation in a Web Farm where multiple servers operate on the same content store is not explicitly supported, and therefore not a test requirement. That said, all content updates cause a shared file to be updated. Whenever this file's time stamp changes, indicating an update to the file's content, the internal caches are discarded and incrementally reloaded and clustered operation is therefore possible without causing cache coherency problems. Both failover clustering and load balancing could be implemented by pointing the Website's storage directory to a network drive, but support for this is, as explained, not a primary concern.

A more central concern is to design an appropriate threading model to handle work inside the engine. Of course, the main purpose of the engine is to respond to synchronous HTTP requests for Web service invocations and HTML resources and therefore we can rely on ASP.NET's threading model for the vast majority of the work: Each request is served by a thread that's allocated from the ASP.NET thread pool.

However, there are several activities that the engine can and should perform asynchronously and in background in order to maximize request/response performance. Because the asynchronous actions differ slightly in their execution characteristics, dasBlog employs three variants of thread use:

The first thread model is in the Mail-To-Weblog feature and another feature called the Xml Storage System update service (XSS), which we have not discussed in this article because it mainly serves to provide an easier migration path for users switching to dasBlog from Radio Userland, both share similar characteristics. The threads for these two features perform periodic actions. Mail-To-Weblog periodically polls a POP3 account and the XSS thread periodically pushes a static copy of the current RSS document to a remote Web service that fronts a distributed file storage system. The only difference between them is that the XSS thread can be explicitly triggered by signalling an event so that content updates are quickly synchronized into the remote store. Neither of these threads is time critical and therefore both run with a below-normal thread priority, which means that they only get served when the system is not busy. Both threads are designed to start up immediately when the application domain is started and to spin infinitely as long the application domain runs. Both threads are robust against internal failures and will fail out gracefully when the application domain shuts down and tears down all running background threads. Except for the signalling event of the XSS thread, the main application does not actively communicate with either thread, but the threads rather invoke functions out of themselves.

The second, slightly different model for threads is employed for handling incoming trackings (that is referrals through, Trackback, and Pingback). Incoming trackings are important to log, but there is no urgency to get them into the data store. One somewhat critical factor relating to detailed tracking of referrers is that it results in a write operation for every incoming request, which creates concurrency issues on the files. Because the required locking results in sequential access to the file store, all trackings are written to an in-memory queue watched by a single thread that runs for the entire application domain lifespan and that processes all trackings in sequence, eliminating concurrency issues. The same strategy is used for sending notification emails to the administrator. While this technique conveniently decouples foreground and background tasks, the primary motivation of this model is to serialize access to limited resources.

The third usage scenarios for threads are Pingbacks, Trackbacks and change notifications that dasBlog actively sends to other sites. Serialization of requests is not required here, because notifications are usually targeted at several different sites, but decoupling from the main thread is imperative because sites might be very slow or unreachable, causing delays of several seconds that can quickly add up when a series of automated Pingbacks needs to be issued based on a single post. Doing this in the thread processing the request (submitting a new or changed post) would mean that the response is delayed by the cumulative time needed for the notifications, which is clearly unacceptable. Therefore, the data that is required to execute these

> "Another concern and requirement for dasBlog was to have good support for localization."

external notifications is packaged into jobs that are submitted to the .NET thread pool for execution.

Using the .NET thread pool permits concurrent execution once threads become available for servicing, but does not create undue stress by recreating new threads all the time. Instead, the active threads in the pool are being reused. This thread pool is shared with the ASP.NET runtime, which by itself puts a global throttle on the threads that can be executed concurrently, limiting the potential to overstress the machine. At the same time, this approach creates a limited risk of drying up the ASP.NET thread pool and causing external requests to be queued up – at the extreme it may even cause the ASP.NET application domain to recycle (shut down and restart). As in many application designs, this is a case where the advantages (easy programming model) must be weighed against the disadvantages and risks. The special use-case here does justify the chosen model, but if these actions weren't confined to the rare case of updating and adding content, the model used for handling incoming trackings described earlier would be a better choice.

Summary

Weblogs are an extremely promising, but much hyped new phenomenon. There are overly enthusiastic claims that Weblogs will change and shake the foundations of journalism and even democracy. While these ideas might dramatically overstate their importance, Weblogs present some real advantages and opportunities for collaboration and a quick and easy way to publish content in an organized manner.

However, as we have shown, creating a Weblog engine that is able to act as a node in an ever changing, growing, distributed, and cross-platform application network comes with a few architectural challenges, even if it the resulting application is relatively small. It must be easy to deploy, easy to use, and easy to customize, navigation must be simple, intuitive and responsive and it must be able to interact and integrate with a wide variety of systems across many platforms. Weblogs are proof that the first generation of Web services is working and that deep interaction between foreign systems is not just a vision of the future, but today's reality – deployed across thousands of servers running Unix, Linux, MacOS X, Windows and many more platforms. They are also proof that commitment to make integration work across all technology camps in a concerted grassroots effort does indeed yield tangible results. None of the Weblog technologies highlighted in this article has ever seen a formal standards committee.

Resources

An installable version of dasBlog and the source code of the latest release can be downloaded from the project's documentation website at http://www.dasblog.net

Clemens Vasters
clemensv@newtelligence.com

Clemens is co-founder and executive team member of newtelligence AG, a developer services company headquartered in Germany. He is a Microsoft Regional Director for Germany. A well-known developer and architect trainer, he is a popular conference speaker, author/co-author of several books, and maintains a widely read and frequently referenced Weblog focused on architectural and development topics at http://staff.newtelligence.com/ clemensv

JOURNAL1

Executive Editor
& Program Manager
Arvindra Sehmi
Architect, Developer and Platform Evangelism Group, Microsoft EMEA

Managing Editor
Graeme Malcolm
Principal Technologist,
Content Master Ltd

Project Management
Content Master Ltd
www.contentmaster.com

Design Direction
venturethree, London
www.venturethree.com

Orchestration
Devinia Hudson
Projects Coordinator, Developer and Platform Evangelism Group, Microsoft EMEA

Foreword Contributor
Simon Brown
General Manager, Developer and Platform Evangelism Group, Microsoft EMEA

The information contained in this Architects Journal ('Journal') is for information purposes only. The material in the Journal does not constitute the opinion of Microsoft or Microsoft's advice and you should not rely on any material in this Journal without seeking independent advice. Microsoft does not make any warranty or representation as to the accuracy or fitness for purpose of any material in this Journal and in no event does Microsoft accept liability of any description, including liability for negligence (except for personal injury or death), for any damages or losses (including, without limitation, loss of business, revenue, profits, or consequential loss) whatsoever resulting from use of this Journal. The Journal may contain technical inaccuracies and typographical errors. The Journal may be updated from time to time and may at times be out of date. Microsoft accepts no responsibility for keeping the information in this Journal up to date or liability for any failure to do so. This Journal contains material submitted and created by third parties. To the maximum extent permitted by applicable law, Microsoft excludes all liability for any illegality arising from or error, omission or inaccuracy in this Journal and Microsoft takes no responsibility for such third party material.