JOURNAL2

Dear Architect

Over the years we have seen a number of transitions in Enterprise computing; from the mainframe model to client-server computing and then to browser-based architectures of the Internet. We are now in the midst of another inflexion point as we move to services-based computing and Service Oriented Architectures (SOA). With every transition there has been uncertainty, discussion, debate and best practice. There have been followers and leaders, winners and losers, systems that worked and systems that failed; this time will be no different!

One thing that has been constant through all these changes is the need for thoughtful, informed and experience-based opinion and guidance from real practitioners. As we navigate our way through the hype, lofty claims and avalanche of press releases looking for real knowledge and real-world experience we need all the help we can get to mine those nuggets of wisdom which will show us how to build real systems.

In this second issue of JOURNAL a model for architectural thinking based on an urban metaphor sets the scene for a great collection of high quality papers where the authors cover a wide range of topics; taking us from considerations for SOA architecture and design, SOA implementation challenges, to the messaging and business patterns required for effective development of SOAs.

As the amount of information on architecture continues to grow the need for real knowledge and experience sharing becomes even more important. JOURNAL brings you some of that experience from fellow architects who are leading the way into the services world.

Enjoy!

Mike Platt
Architect, Developer and Platform Evangelism Group, Microsoft Ltd

Editorial

By Arvindra Sehmi

Dear Architect
Welcome to the spring issue of JOURNAL. With this second issue we've gone global. This is a direct result of the tremendous feedback for JOURNAL1 from Microsoft's customers and partners, together with encouragement from colleagues worldwide. For that incredible vote of support I'd like to say 'Thank You'!

I am proud to have put together a team that believes unfailingly in the concept of a dedicated publication for IT architects and to have the support of some of the most 'agile' senior management in the business. With new sponsors, like Harry Pierson in Redmond, we will take this publication to a bigger audience and extend its reach through multiple delivery channels, including the web, digital download, and through MSDN Library which becomes the long-term 'store' for JOURNAL. We will also reformat JOURNAL to support individual printing and easier on-screen reading.

Meet Harry; he's a smart thinker in the Platform Strategy and Partner Group in Microsoft, Redmond. He is determined to make JOURNAL available through as many channels as he can dream of, within limits of course. Harry has numerous ideas on how to widely communicate the writing of our authors who are first and foremost the soul of this publication. As editors our mission is simple – make sure our authors get heard. So JOURNAL is gearing up for massive reach and the ability to influence thinking about architecture, past, present and future.

We are fortunate to open this issue of JOURNAL with Pat Helland's first public written paper on Metropolis, which uses analogy and metaphor to explore the present and future directions of IT by studying the recent history of our urban centers. Pat's reputation is formidable and as the Architect lead for COM+ and SQL Server Broker past experience tells us that if he has something to say it's probably worth listening to. Metropolis is part of his ambition to write a book on SOA and 'autonomous computing' therefore we expect more papers from Pat will appear in future issues of JOURNAL as his ideas make their way from incubation to ink.

Lawrence Wilkes and Richard Veryard, from their think tank CBDI Forum, remind us that for all the hype surrounding SOA we must not forget that the objective is to build agile systems in support of the business. Sometimes we get so carried away with the compelling power of emerging Web services technology that we need to be brought back to earth again. Their list of principles and best practices does just that.

Easwaran Nadhan from EDS demonstrates how companies must progressively construct components and services involved in the implementation of SOA. He postulates that a road map and company-specific standards are key prerequisites ensuring systematic implementation of such enterprise wide architectures. He identifies eight key challenges a company faces in SOA implementation and uses real-world examples to address these challenges.

Microsoft's Philip Teale and Robert Jarvis of SA Ltd introduce the first part of a paper discussing business patterns defining architectural templates for business solutions. They identify a set of architectural elements required to fully describe business patterns. This set has been classified and focuses on elements that describe the most stable parts of a business suitable for subsequent 'patternisation'. Part two of this paper will appear in the next issue of JOURNAL.

Soumen Chatterjee from CGE&Y gives us a description of messaging patterns in SOA. Traditionally messaging patterns have been applied to enterprise application integration solutions, but Soumen uses these patterns to explain how a SOA can be implemented. His insight shows us that messaging patterns can be applied equally well at the application architecture level, especially in SOA-based solutions, because they too are fundamentally message-oriented. See the next issue of JOURNAL for part two.

Please keep up to date on the Web at the Microsoft® architecture center and specifically at the new home for JOURNAL http://msdn.microsoft.com/architecture/journal where you'll be able to download the articles for your added convenience. If you're interested in writing for JOURNAL please send me a brief outline of your topic and your resume to asehmi@microsoft.com.

Now immerse yourself in this issue's fascinating world of thoughts, ideas and sheer good advice from some of the world's leading architects.

Happy reading!

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

Keep updated with additional information
at http://msdn.microsoft.com/architecture/journal

Metropolis

By Pat Helland[1], Microsoft Corporation

A metaphor for the evolution of information technology into the world of service oriented architectures.

This paper explores the idea that information technology is evolving in a fashion similar to how American cities have evolved over the last two centuries. The opportunities and pressures of the technological revolution have driven our metropolises to adopt new frameworks, models, and patterns for commerce and communication. Recent developments in IT are analogous. What can we learn about the present and future directions of IT by studying the recent history of our urban centers?

Introduction

In this paper, we are going to explore how there are similarities between the evolution of cities in the 19th and 20th centuries and the development of IT shops. In both cases, we saw gradual evolution of environments that developed in isolation. This independent development resulted in many differences of culture and how things were done.

In the middle of the 19th century, the railroads connected the majority of the cities in the United States. This resulted in people and stuff moving around in a fashion not previously possible. Moving people and stuff provided the impetus for ensuring the stuff worked together and met standards of compatibility. The changes in expectations and capabilities fueled an explosion in retailing, manufacturing, and led to the urbanization of American life.

For many years, IT shops have developed in isolation with independently developed applications and little overlap in how things were done. Since the applications weren't connected, this seemed of little consequence. Recently, it has become very practical to interconnect both the applications inside an IT shop and multiple IT shops spread around the world. People can now easily browse and visit distant applications. Chunks of data are easily transmitted to remote applications. What is still difficult is to make the data work across different applications.

Metropolis is an examination of this analogy in an attempt to both explain what is happening in Information Technology today and to show us what we can expect to happen in the future. There are eight facets to this analogy that we explore:

– *Cities* map to *IT shops* – these are systems of systems, isolated from each, trying to cope with the arrival of the railroad.
– *Factories or Buildings* map to *Applications* – the broad-stroke componentization of the isolated systems.
– *Transportation* maps to *Communication* – the impact of the railroad being analogous to the impact of the Internet.
– *Manufactured Goods* map to *Structured Data* – each changing with the arrival of standards that are so disruptive to the existing custom wares.
– *Manufactured Assemblies* map to *Virtual Enterprises* – as bicycles became assemblies of best-of-breed components, so business processes will become pipelines of best-of-breed service providers.
– *Retail and Distribution* maps to *Business Process* – interoperable metadata, such as standard sizes and ingredients lists, permit transformations and compositions.
– *Urban Infrastructure* maps to *IT Infrastructure* – as the city grows, economic pressure mounts to develop common services, such as water, sewer, and road maintenance.
– *City Government* maps to *IT Governance* – both experience the pressure to balance investment in functional growth with investment in infrastructure.

Our cities transformed themselves from isolated, quirky heterogeneous systems to highly inter-operable systems in less than a century. Let us see what we can learn from their experience on our own path to streamlined interoperability!

Cities ↔ IT Shops

Cities gradually evolved as sites to gather for both commerce and manufacturing. Inside the cities there were independent buildings with little or no connection between them. You might consider placement with respect to the road, but that was about the extent of the relationship to other buildings in the city.

[1] The author gratefully acknowledges Mike Burner's assistance in preparing this paper. Also, the artistic contributions of Emmanuel Athans have added tremendously and are deeply appreciated.

Most of us forget today that it was a hard day's ride on horseback to go 100 kilometers. Only the most hardy and adventurous would travel that far in their lifetime. Cities had only limited exposure to each other and developed their own culture, style, and way of doing things. Similarly, IT shops gradually evolved as new applications were built and then extended and evolved. Each application stood apart and independent of its neighbors in the same IT shop. Each IT shop had its own culture, style and way of doing things. Only the most hardy and adventurous traveler would visit multiple IT shops.

Economic pressures changed our cities. Certainly the best intentions of city planners eased the transitions – and saved some historic monuments – but economic opportunity is what really drove cities to modernize, to share services, and to devise creative means to achieve efficiencies.

Economic pressures are changing our IT shops, with or without master planning. As you build new applications and 'renovate' old ones, you must consider how to link them to the shared infrastructure. How will they be connected? How can they leverage common information architecture? How will they be factored to maximize reusability? These are the challenges and trade-offs we all need to consider.

Factories and Buildings ↔ Applications

In the early part of the nineteenth century, manufacturing was typically simple and independent. The goods that were produced were limited by both the appetites of the local market and the sophistication of the manufacturing process. Factories were largely vertically integrated, producing all of the parts of the final assembly, assembling them, and even selling them. If you wanted boots you might go to the tannery. This was not the most efficient approach to the creation of goods and the manufactured items were expensive and usually not of the highest quality.

Most of our applications today are like those tanneries of the early 1800s. They produce processed data independent of each other, and deliver into limited 'markets'. They are vertically integrated and don't very often accept the work of other applications as input.

The railroad profoundly altered manufacturing. By driving down the cost of transporting manufactured parts, transportation permitted local manufacturers to produce higher quality, more sophisticated goods. Now the boots from the tannery had steel eyelets to keep the boots from tearing at the laces, and woven laces that held up better than leather thongs. Componentization allowed artisans to focus on their core competencies, rather than have to understand the diverse processes necessary to produce all of a sophisticated assembly.

Moreover, transportation made new markets available to businesses, allowing them to grow and specialize. A talented potter might come to dominate a regional market by shipping her wares up and down the rail line. ISVs – independent stuff vendors – proliferated.

For both factories and applications, independence is essential. You simply can't get any work done if you need to get everything to work together perfectly. It is only by decoupling the

"Our standards efforts are just beginning. Most application integration today is done by people, expensively and with high rates of error. Information integration should be a focal point for IT, since it offers huge opportunity for returns on investment."

evolution of these pieces that you can achieve change. Yet there are inescapable advantages to interconnection. You can leverage the work of others (factories or applications) to accomplish your work. The demand of others for your work provides the economic stimulus that gives you a reason for existing.

Transportation ↔ Communication
In the middle of the 19th century, the railroad arrived! Tremendous amounts of money were made moving people, coal, and wheat. It was the delivery of people and the basic

(non-manufactured) goods that drove the creation of the rails. Incredible booms and busts occurred in the 1840s and 1850s as speculators built short lines connecting a couple of cities. Eventually, these were consolidated and standards were established to allow nationwide connection by rail.

The movement of people by rail stimulated tremendous changes. People traveled to strange and wondrous places! In addition to the sweeping cultural changes brought about by travel, retail began to expand dramatically as people hopped the train to go buy stuff from other cities. Retailers were now able to gather goods into stores to offer it for sale in new ways. The movement of stuff also caused change as there were new expectations that things would work together. Before railroads it simply didn't matter if one manufacturer's goods were incompatible with another manufacturer's goods.

At the end of the 20th century, the Internet arrived! Tremendous amounts of money were made in browsing, email, jpegs, mp3s, and chat. The wires were laid to provide browsing and the movement of the simplest forms of data. The .COM boom and bust are now legendary.

People browsing changed things. The browser allowed a person to be transported to directly interact with a distant application. This direct access has driven demand for sophisticated business processing as people question why automated interaction can't replace their direct interaction via browsers. The changes from the movement of data are just beginning, though. Data still does not work together well and automated business process is still very limited.

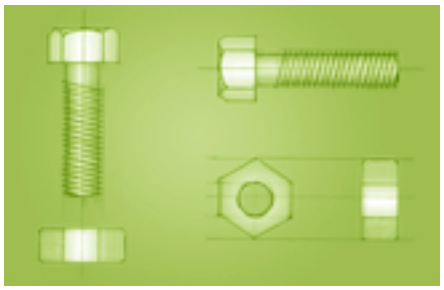
Both transportation and communication started by moving people and the basic commodities (either commodity goods or data with simple structures). These new connections drove new changes in the standardization of stuff and data. They drove changes in retail and (soon) in business process.

Manufactured Goods ↔ Structured Data
In the early 1800s, goods were hand-crafted. Assemblies were created with 'trim and shim'; if the bolt of a lock didn't quite fit the slide, you trimmed a bit off with a file; if the bolt rattled in the catch, you shimmed the catch to get a tighter fit.

Pioneers like Honore LeBlanc and Eli Whitney introduced the idea of standardized parts into the manufacturing process. By establishing tight controls over the specification and production of component parts, Whitney was able to produce 12,000 muskets for the US military, effectively freeing the United States of its dependence on overseas craftsmen for its military weapons. But this was still 'in house' standardization: all of the parts were produced in one place, under one roof, and with one set of controls.

By the late 1800s, the idea had expanded across manufacturers, and de facto standards had emerged for common parts. There were sizes for nuts and bolts and cylinders, with the expectation that the ones produced by one factory would be interchangeable and interoperable with similar and complementary components produced by another. Companies that produced parts with a high degree of precision thrived; those with less consistent processes failed.

Today we still have mostly non-standard data structures. Every application models information its own way, and we depend on human operators to 'trim and shim' to integrate the applications. We see the beginnings of this movement in the XML and Web services specifications. We now have language syntax and some rudimentary rules for exchanging structured data.



Moving forward, we need to add semantics to our cross-application understanding. This will occur both in horizontal and vertical sectors. Just as the marketplace demanded interchangeability of goods in the late 1800s, it will demand the interchangeability of data in the near future. This interchangeability will be at the level of the semantics for real business-level interaction. This means standardizing the functionality of business concepts such as customer and purchase order. Data items that will be shared across applications need standardization.

Applications must retool or perish. Organizations that fail to realize the efficiencies of integration-by-design will lose in the long run to those who pursue them. The efficiencies resulting from these changes will be an economic windfall to the companies that survive the transition! Just as manufacturing standards have dramatically improved our lives, the effective use of business-level standards for data will dramatically improve our lives.

Manufactured Assemblies ↔ Virtual Enterprises

Most bicycle manufacturers do not produce tires, just as dressmakers do not manufacture their own eyelets. By creating assemblies out of best-of-breed components, individual bike makers can produce higher quality, more sophisticated products. Competition among the component manufacturers drives efficiencies and quality improvements. Bikes just keep getting better.

To do this, they need detailed specifications for the component parts. You have to match the width of the tire with the wheel with the fork with the brakes with the axle bolt. You have to consider the context in which the part will be used. Is weight or ruggedness the principle concern? Manufacturing became a value chain driven by information, reputation, and trust. Companies partnered to change the process of bringing goods to market.

Today, companies are 'creating assemblies' of their business functionality. Rather than create a distribution and shipping department, the work is outsourced. Rather than actually building the product owned by the company, its creation is outsourced to a company that specializes in low cost, high quality manufacturing. The engineering, marketing, and ownership of the product may remain in-house (or may not remain in-house). The definition of a 'company' is evolving as surely as the definition of an assembly did in the last century.

High-speed communications and structured information offer the same promise for many other business functions, giving rise to the 'virtualization' of our organizations. A business component model can be created by clearly defining the semantics and operational requirements of our business capabilities. With clear interface definitions, we can encapsulate the details of how these capabilities are implemented. Business process becomes a traversal of these component capabilities, so each component can

be orchestrated as a member of any number of processes, and may be transparently relocated inside or outside of the organization.

Value chains can be created and recreated using best-of-breed business capability providers. The 'owner' of a business process might do little more than orchestrate it. Marketing, sales, manufacturing, distribution, legal services, and human resources support might all come from specialized business service providers. The best result is agility and competitive flexibility. If distribution is backed up you just engage additional logistics providers; if your accounting service is unresponsive you replace them.

To do this, you need detailed specifications for the component capabilities. You have to match the volume of goods shipped with the countries into which you sell with the delivery guarantees consistent with the service levels your customers demand. You have to consider the context in which the capability will be used: is security or transparency the principle concern?

Standards allow the composition of stuff. Better stuff is created, because component providers can leverage the cost of optimization across a broader market. Competition drives increased efficiencies, as does a sharper focus on the unique competencies of each provider in the value chain. Business process will just keep getting better.

Retail and Distribution ↔ Business Process

By the late 19th century, urban centers had developed significant retail districts. Goods had gotten more sophisticated and consumer choice had improved, but shopping was still quite an expedition. Shopping day might include taking a train into town, and then going from butcher to baker to greengrocer to dressmaker to milliner to cobbler to jeweler in search of everything you needed that week.

Stores were often still the front rooms of the factory. Visiting the shoe store might receive a request to return in hour because your shoes were not quite ready. Much of what was purchased was still custom made, and therefore quite expensive.

The new distribution capabilities enabled new approaches to retail. The standardization of sizes significantly reduced the cost of many goods by permitting mass production (while at the same time encouraging people to wear ill-fitting clothes)[2]. And the ability to transport goods to central locations for sale gave rise to the department store and the supermarket.

And then along came Wal-Mart! Wal-Mart achieved new efficiencies by wielding the power of retail over the manufacturers. Wal-Mart set the standards, not the manufacturers; manufacturers complied or Wal-Mart did not carry their goods. The effectiveness with which Wal-Mart delivered pleasant, low-cost, one-stop shopping made the store a destination for many shoppers; the power had effectively shifted to the retailer.

Now let's examine the state of the art for business process. A major innovation is *swivel chair integration*. Today's leading form of B2B computing is known as *fax-and-pray integration*.

An important technique for reducing integration errors is called *ALT-TAB integration* which allows the use of the clipboard to copy data between applications.

If we are to do better, we need *interchangeability* of our data and operations. Standardized and interchangeable clothing allowed for inexpensive production of the clothes. Furthermore, this approach allowed the separation of retail and distribution from the creation of the goods. For us to make major inroads in business process we need standardized and interchangeable data and operations to allow for better composition. Then we can create computing resources and pre-allocate them for later use. This technique allows the manipulation of the resources to be handled separately from the creation of the resources. This is an essential requirement to the creation of a separate mechanism specifically for business process.

We have seen an amazing transformation in retail. People cheerfully accept standard stuff and customization is rare and expensive. But business process is still largely hand-crafted. There are poor standards. It involves a lot of 'trim-and-shim'. We have very poor 'interchangeability'.

[2] Most of us forget that standardized clothing was a prerequisite for daily bathing. Few could afford more than one custom-made suit and it didn't make sense to bath if your clothes were going to stink, anyway!

Finally, it is clear that business process will grow to be the driving force that dictates the shape and form of applications. The work is done via business process. This will become more malleable and separate from the resources being managed. As business process becomes the economic driver, it will dictate the shape, form, and standards of applications as surely as Wal-Mart drives the standards for many, many manufactured goods!

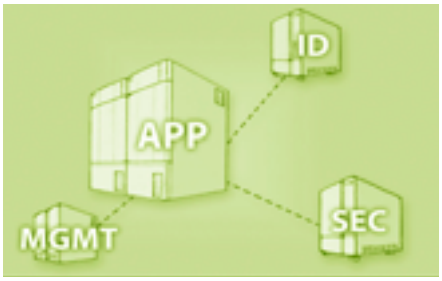
Urban Infrastructure ↔ IT Infrastructure

By the late nineteenth century, the cities had grown and lots of people were living in them. Pretty soon … it smelled bad. Urban density drove urban infrastructure. Common services such as water, sewer, gas, electricity, and telephony were built or licensed by city governments to achieve efficiencies, realize opportunities, and make the city a more livable place.

These efforts required metropolitan services like dams, power plants, and sewage treatment plants (or perhaps just sewage pipes run into the bay). In addition to the metropolitan services for the infrastructure, you also needed to hook the services up to every building in the city. Running the services to the buildings was the first problem; connecting to legacy buildings was frequently a nightmare.

Many buildings were retrofitted. The Cathedral Notre Dame de Paris has flush toilets and electric lights. They were not installed when the structure was first built. The pace of change in building technologies over the last fifty years has taught us a lesson: now we put conduits from the street to the building to anticipate evolving cabling requirements.

Building and connecting to infrastructure services that are shared is usually a mix of public and private funding. Huge infrastructure projects are usually funded by the city, but individual connections are usually paid for by the businesses and homeowners. Major new private developments may not be approved unless they commit to paying for infrastructure improvements. For example, a new shopping center might have to pay for road improvements in front of the complex.



IT shops have grown and lots of applications are living in them. Some IT shops … smell bad. Business process owners need to re-authenticate application to application. Process ownership roles are hard to map to application-specific authorization schemes. Dozens of applications produce dozens of logs in dozens of formats. Exceptions can be hard to route, parse, or use to take action. Process data is dispersed across many application-specific databases, inhibiting transparency and data mining.

Common services, such as authentication, authorization, logging, naming, and directories need to be commissioned by organizational governments, and applications need to be built or retrofitted to use them. This may require metropolitan support: without funding and a mandate from senior management, the investment is likely to be beyond what an individual process owner can afford.

Once the mandate and common services are in place, operational compliance will be required and paid for by every new application. Ease of operational integration will become a key criterion in the selection of packaged applications.

Crowded environments need well-designed infrastructure services to function smoothly. The cost can appear daunting in the short term, but the long-term cost of not doing it may drive you out of business. Just as with metropolitan infrastructure, there is competition for funding between the goals for new business functionality and the infrastructure needed to make the whole mess work well.

"Business process modeling and orchestration is nascent. It will grow to become the dominant force in how applications are designed and integrated. Today, most understanding of organizational business processes is in the heads of the people that own the process."

City Government ↔ IT Governance
Cities have different visions for their shape and form. To realize their visions, governments engage in city planning. Seattle envisioned itself as a world-class city, and so engaged in bold reinventions of itself, from flattening Denny Hill, to linking the lakes to the sea, to building the space needle for the 1962 World's Fair, to spanning Lake Washington with bridges that usually float. Other cities take the opposite approach, using city planning to limit growth in order to protect livability.

To implement their vision, cities typically enlist the cooperation of both business and government. Usually, businesses are the drivers of the growth and municipalities constrain and direct the growth to match their vision.

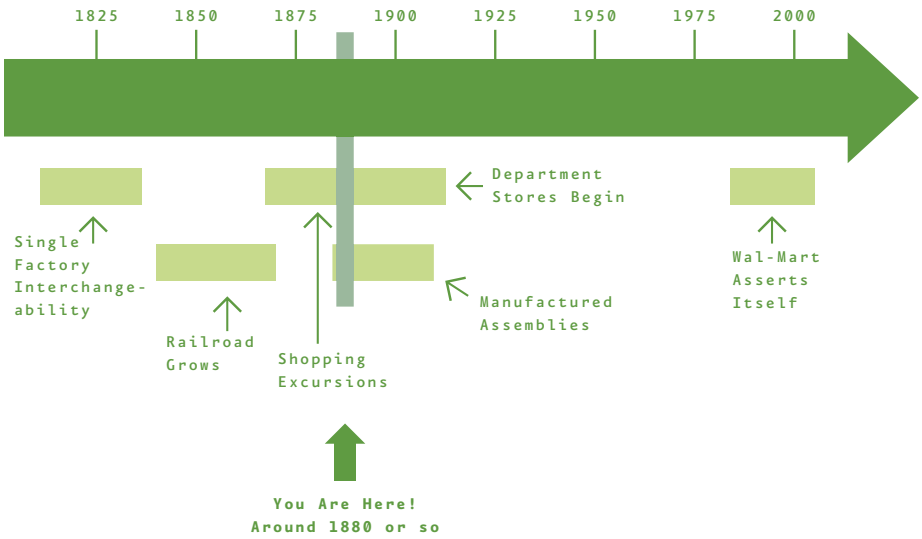
Different visions lead to different infrastructure goals. To encourage growth, cities need to overbuild infrastructure capacity. Care must be taken to balance infrastructure investments! If a city actively pursues growth but fails to anticipate the impact on transportation, for example, congestion and inefficiency will result (come visit Seattle if you have any doubts). Zoning, available infrastructure, and business incentives drive investment in private industry. Trade-offs are made in public/private funding according to the perceived value to the community. An easy way to start a civic debate is to propose a new sports stadium be built using public funds.

So city governments must allocate resources across an array of competing priorities, taking care to protect the sacred – such as education – and balancing the short-term, long-term, and speculative. Private industry makes decisions about what buildings to erect; constrained and controlled by planning rules.

IT governance is not so mature. Who makes the tough choices in IT? Is it the CEO, the CIO, the business unit leaders, techies, or perhaps committees? Are priorities established based on cost, flexibility, or asset utilization? What is success, and how is it measured? Are we seeking cost reductions, business process transparency, or competitive advantage?

Enterprises might learn a lot by looking at how cities manage the difficult process of resource allocation. What proposals are projected to pay for themselves? What is the timeframe and risk analysis around those projections? What in your organization is sacred? What resources remain after funding those efforts? What balance of short-term, long-term, and speculative investments are right within the specific corporate culture?
These problems are common for metropolitan and IT environments.

It is interesting to note that most American cities allocate resources in a way that is optimized for growth. The decision to build is usually in the hands of the business and this optimizes for growth over cost management.



"IT investment is a balance of funding the sacred, protecting historic monuments, and allocating spending between infrastructure and business opportunity. Striking this balance is a key facet in effective governance."

Looking to the Future
So where do we sit in this timeline when we compare urban development with IT shop development?

We've seen the usage of sophisticated data structures that interchange within a single application. This is analogous to *single-factory interchangeability*. We've seen the Internet connect IT shops and the applications within those shops similar to the *railroad's growth*. It is common-place to browse a strange and distant application just as our grandparents hopped the train for a *shopping excursion* and visited many different stores in a distant city. Virtual enterprises (analogous to *manufactured assemblies*) have barely begun. We are at the cusp of succeeding in simple business process (analogous to *department stores*). We are at approximately the early 1880s in urban development in our parallel IT shop's growth!

We can see from this the innovations ahead lie in the creation of standards and interchangeability. These will allow the interconnection of inter-operable pieces of computing that can move into these different roles. This will allow tremendous growth in business process and, as that business process spans companies, the increasing growth of virtual enterprises.

It is this need for independent, and yet inter-operable, pieces that leads us to the service-oriented architecture and the changes we see beginning in application architecture. This is not to suggest that it will take 100 more years to see a Wal-Mart equivalent, but we will the economic forces driving us to the dominance of business process over application and service standards.

Conclusion
We have a fun time ahead of us. We are building boom towns with no end in sight. Sure, it won't take a hundred years for us to get there, but we won't be done tomorrow, either. The same forces that drove the maturation and sophistication of cities, civic infrastructure, and business models are driving IT today.

What do we have to look out for? How do we prepare for this adventure?

Remember that heterogeneity happens. Unless you have a very simple application portfolio, shared services will not be achieved by trying to put all of your applications on one version of one platform. Even if you could, the next merger would change that! Rather, you have to design for interoperability and integration across platforms. This is the force that is driving the industry wide work in service oriented architectures.

IT investment is a balance of funding the sacred, protecting historic monuments, and allocating spending between infrastructure and business opportunity. Striking this balance is a key facet in effective governance, and in realizing the potential of IT in your organization.

Our standards efforts are just beginning. Most integration today is done by people. It is expensive and has high rates of error. Information integration should be a focal point for IT, since it offers such tremendous opportunity for returns on investment. Benefits will be realized through cost reduction, error elimination, more effective customer interactions, and generally better business intelligence.

Business process modeling and orchestration, too, is nascent, but will grow to become a dominant force in how applications are designed and integrated. Most understanding of organizational business processes is in the heads of the process owners. Tools and techniques for capturing and refining these processes will greatly enhance the productivity of those process owners.

You have to maintain a light hand. It is counter-productive to try to dictate what happens in every structure in town, what color shirts are made, and how much is charged for soap. You have to embrace the semi-autonomous approach to governance that is characteristic of our cities, and allow the process owners to optimize and achieve efficiencies with as few constraints as possible.

Enterprise application portfolios will undergo a significant refactoring process to embrace a model that allows more autonomous control of business capabilities. Effective modeling and a light hand on the tiller will permit dynamic, distributed organizations to create and deliver more value than ever before. In ten years time, IT shops will evolve from looking like the cities of the 1880s, to looking much more like the cities of today.

Just remember to invest in the transportation systems!

Pat Helland,
Architect, Microsoft Corporation
phelland@microsoft.com
Pat Helland has 25 years of experience in the software industry and has been an architect at Microsoft since 1994. He has worked for more than 20 years in database, transaction processing, distributed systems, as well as fault tolerant and scalable systems. Pat worked at Tandem Computers designed TMF (Transaction Monitoring Facility). He was one of the founders of the team that implemented and shipped Microsoft Transaction Server (MTS), now COM+. Pat has recently focused his thinking on loosely-coupled application environments.

Service Oriented Architecture – Considerations for Agile Systems

By Lawrence Wilkes and Richard Veryard, CBDI Forum

When designing business software, we should remind ourselves that the objective is delivering agile systems in support of the business; not Service Orientation (SO). Rather, SO is the approach by which we can enable business and technology agility, and is not an end in itself. This must particularly be borne in mind with references to Web services. Achieving the agility that so often accompanies Web services is not just a consequence of adopting Web service protocols in the deployment of systems, but also of following good design principles. In this article, we consider several principles of good service architecture and design from the perspective of their impact on agility and adaptability.

Web services provide a powerful framework by which we can deliver more agile solutions. However, we must combine their use with principles of Service Orientation that ensure that agility requirements are met. We can see parallels with the adoption of component technologies and Component Based Development (CBD). Components promised benefits such as reuse and an open market in components, which together would drastically reduce the time to deliver new systems. However, while component technologies like Microsoft COM have been widely adopted, most organisations saw little reuse and the open market in components failed to grow to anywhere near the level predicted. IT did get other benefits from componentisation of course; such as improvements in system scalability and the ability to replace components as needed, but many of the claims we saw being used to justify investment in componentization were not realised.

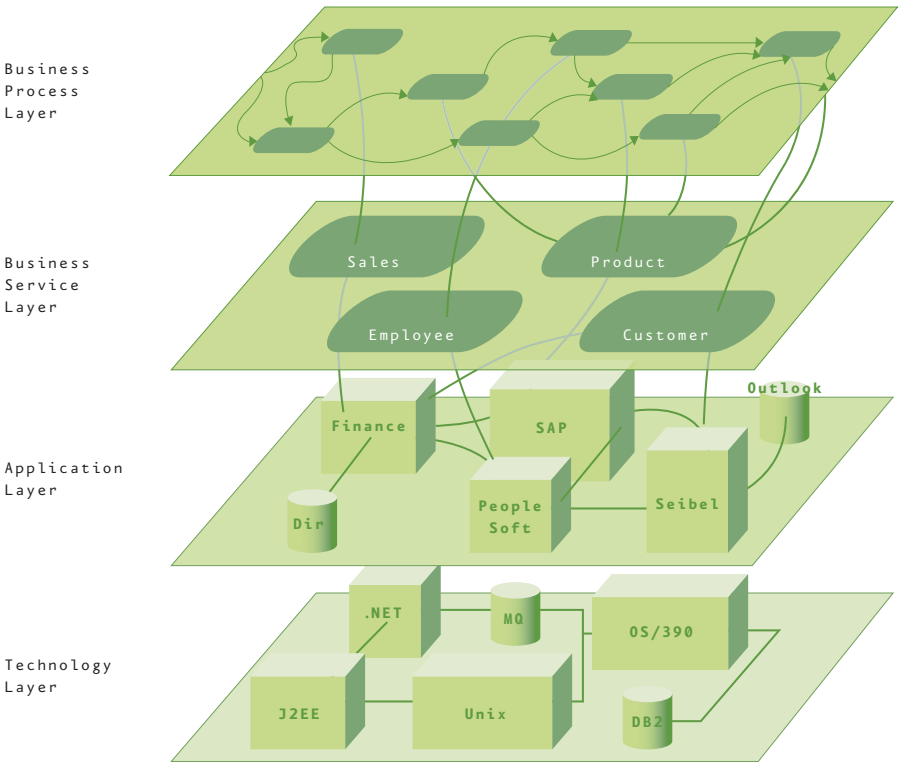
Why was this? Well, while component technology was a great framework for reuse, developers didn't put the effort into making sure the components themselves were actually designed for reuse by another project. Typically, the effort needed to understand what an existing component did and adapt it for a new requirement outweighed the effort to simply build a new one that fitted the requirement perfectly from scratch; except of course that consequently the new component wasn't reusable by the next project either, and so the cycle continued. In other words, while many organizations adopted component technology, they didn't adhere to the principles of CBD.

We now face a similar challenge with services. We can predict that Web service technology will be widely adopted, but to what extent will they be based on the principles of SO? And if they are not, will the promise of Web services be realised – particularly for the business?

Loose Coupling

Loose Coupling is one of the mantras of Web services. No discussion of Web services seems complete without some reference to the advantages of looser coupling of endpoints (applications) facilitated by the use of Web service protocols.

Figure 1. Decoupling applications and technology through services



"While component technology was a great framework for reuse, developers didn't put the effort into making sure the components themselves were actually designed for reuse."

The principle is that by using a resource only via its published service and not by directly addressing the implementation behind it then;

1. Changes to the implementation by the service provider should not affect the service consumer.
2. The service consumer could choose an alternative instance of the same service type (for example change service provider) without modifying their requesting application, apart from the address of the new instance.
3. The service consumer and provider do not have to have the same technologies for the implementation, interface, or integration when Web services are used (though are both bound to use the same Web service protocols).

The concept of coupling applies within the business itself regardless of any IT systems, between the software applications, and at the technology layer. *Figure 1* illustrates that coupling takes place at a number of levels.

At the **Technology Layer** we are concerned with integration at the platform and network level. For example how do you connect J2EE to .NET? Considering integration technologies at this level might involve distributed computing or messaging products.

At the **Application Layer** we consider how applications are connected to other applications; for example connecting Seibel with SAP. Enterprise Application Integration (EAI) technology is the common approach at this level.

Web service protocols may enable loose coupling at the technology layer, but not necessarily in the application layer. For example, use of Web services provided by a packaged application such as SAP might remove technology dependencies in comparison to use of their BAPI interfaces that use component technologies. But the Web services exposed are still the same SAP specific interfaces and are not decoupled from the application. Consequently the service consumer is still tightly bound to SAP regardless of the use of Web services.

We therefore need a separate Business Service Layer that abstracts the service away from both the technology and the application. With this we can then support the business with a Business Process Layer that interoperates at the business service layer with customers and sales (concepts that do not change) and not directly at the application layer with SAP with Seibel (implementations that do change).

However, some of the loose coupling benefits of services and Web services for B2B integration might seem irrelevant if the business has for example entered into a multi-year contract to use a single supplier. Even so, at the end of the contract the business does not want to be held to ransom by a supplier who knows that the tight coupling with their systems is now a constraint on change. This is why we need to design SOA against a business requirement for specific forms of adaptability. You should not assume that all forms of loose coupling at the different levels are automatically valuable to all organizations.

Service Provider and Service Consumer Perspectives

Service agility can be seen from both the perspective of the service provider and the service consumer. *Table 1* considers some of the agility requirements of both and the general service design principle that could satisfy them. The service consumer's real objective is normally to receive maximum value at minimum cost and risk. For a service offered on a commercial basis, risk can be reduced by aligning the payment of services with the benefits received – this should be achieved by some form of pay-per-use but only if the service and its commercial terms are properly designed. Risk is also reduced by the ability to switch service provider to gain tactical or operational improvements in price or service level.

One might expect that the service provider's objectives are a mirror of this – deliver maximum value at minimum cost and risk. Service value is maximized by making the service easily available to as many potential users as possible, in as many use-contexts as possible. Cost is reduced by the economics of scale, as well as having an efficient response to new service demands. Risk is reduced by spreading the service across a wide variety of different uses and contexts, so that demand peaks and troughs are smoothed, and by using virtual (utility or grid-based) resources to handle variations in demand levels. If service consumers each want to use a service in a different way it makes more sense for the provider to try and deliver a generalized service that enables this.

Table 1. Balancing service provider and consumer Needs

Service consumer Objectives	Design Principle
Reduce effort to use new services.	Precise specifications. Standardized services.
Choose alternative instances of services type.	Services well abstracted from implementation. Standardized services.
Service provider Objectives	
Reduce demands from new consumers for additional features.	Coarse grained, abstracted services that meet a wide range of service requests.
Compose New service from existing ones.	Fine grained, generalized services that can be composed in a variety of ways.
Reduce impact of changes to service Implementation.	Services well abstracted from implementation.
Provide service in new and unforeseen context.	Generalized services.
Provide service to as broad a range of consumers as possible.	Coarse grained, and generalized services.

These objectives all have implications of the design of the service from the provider's perspective. However, there also may be a conflict of interest. For example, the service provider might deliberately seek to reduce the agility of the service consumer to use alternatives by locking them in to inflexible service designs.

Design Principles

In this section we examine in more detail some of the design principles that have so far been mentioned in passing such as abstraction, and understand how they contribute to agility.

Abstraction

The principle of *Abstraction* is normally used in the context of ensuring that a service is independent of a specific implementation and other detail. As discussed earlier, Web services provide a high degree of abstraction from the service Implementation by using standards-based protocols rather than the native interfaces of the underlying technology. One of the principles of Service Orientation is to focus on what a service does, not how it does it.

To enable agility, we use abstraction to:

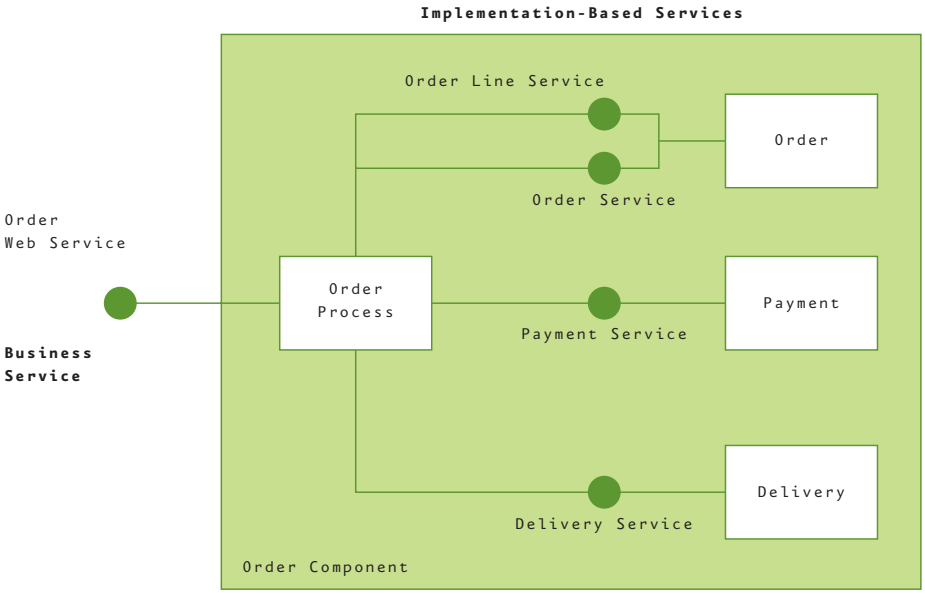
– Remove implementation specific references from the service.
– Hide data or behaviour in the implementation that is specific only to the internal working of the implementation and not important to the service consumer.
– Transform data types that are specific to the implementation technology.
– Hide object behaviour. Unlike object interfaces which might encapsulate the implementation we typically do not want to expose object behaviour in the service that gives rise to inheritance, creates dependency on a specific object technology, or forces the service consumer into using an object oriented approach that may be inappropriate to the messaging style.

Some of these can be seen as similar to principle of encapsulation in object and component technology. However, the goal is not just to hide detail behind an interface, but also abstract that interface away from the implementation technology and any other implementation specific features to reduce coupling.

When building a new component, this abstraction can be built into the services from scratch. The component might offer implementation-based services that are used within the component and the sub-assembly of highly related components around it where tight coupling should not be an issue (though use of Web services protocols might still be desirable for

"One of the principles of Service Orientation is to focus on what a service does, not how it does it."

Figure 2. Component exposes abstracted business service



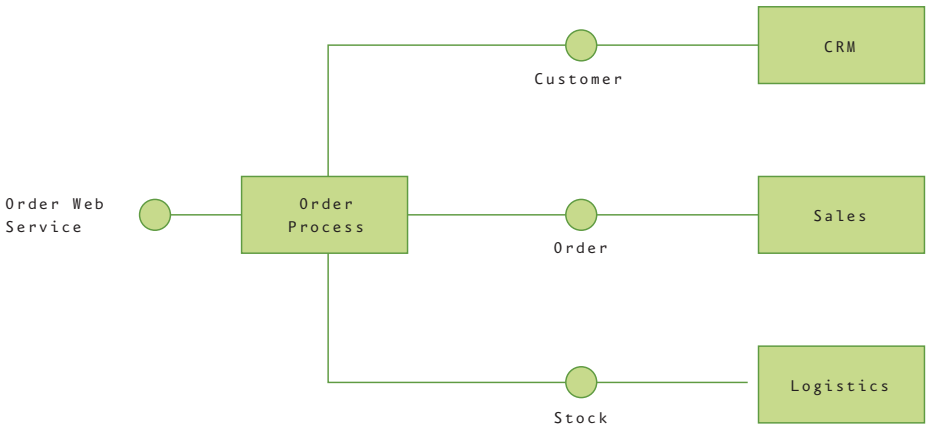
other reasons such as dynamic addressing of the component for scalability), whilst a more abstracted business service is exposed externally. As illustrated in *Figure 2*, a process component is a common way to architect this. The business service exposes operations at the order level, not the individual object level.

The purpose of this is:

1. It means the service consumer does not have to know how to use all of the individual implementation-based services to place an order.
2. From an agility perspective, the way in which the service provider chooses to configure those internal components can change over time without affecting the service consumer.
3. The implementation-based services are still available to other developers in the project who require them to compose other business services.

More often perhaps in the current climate, the business service is going to be implemented by a number of existing applications. As shown in *Figure 3* the same pattern still applies, with a new process component abstracting the business service away from the implementation-based services offered by the packaged applications. Again the implementation-based services are still published so they can be composed into other services for different business requirements. Agility can also be improved not just by abstracting the service away from the underlying implementation but also by taking a more abstract view of the business concepts in the service so that they can be used in a broader context. For example the use of a party information service instead of separate ones for customer information, supplier information, and so on. The benefit being that new party types can be accommodated without having to deliver a new service each time. The difficulty however is that each of these types is likely to have specific data or operations that are not common to the others, which means we also have to consider the use of generalization.

Figure 3. Process Component provides a façade across existing implementation



Generalization

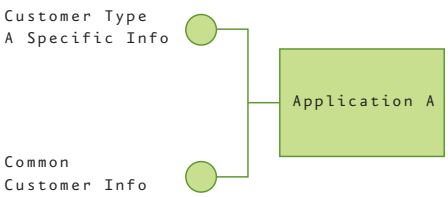
The principle of generalization is to broaden the application of a service so that it can be used in a wider range of scenarios, including unexpected ones, removing the need to build a specific service for each new requirement.

The goals of service generalization to enable agile systems include:

– Separating out common data and behaviour from the specific, so that the common parts are more broadly applicable to a wider range of requirements and composable into many other services, as illustrated in *Figure 4*.
– Including a wider breadth of data and behaviour in the service than some individual service consumers might require, so that the service meets the need of broad range of service consumers without having to deliver them each a different service.

Figure 4.

Generalized service separates common from specific



These first two statements might appear contradictory. However, this reflects a two stage process to deliver agile services. First, decompose the service to find the fine grained parts both common and specialized; and then compose them back together to form a coarse grained service. Clearly this can also be considered an aspect of granularity which is discussed later.

An objective of using generalization to deliver a coarse grained service would be to reduce the number of services that need to be exposed and maintained. It can reduce the maintenance that is necessary when a service consumer want to use the service in a new context, or realizes that they need some additional information from the service that they have not used before. With foresight, the generalized service can already provide this.

Depending on the particular requirement, generalization might be at the operation or the message/document level. It could be two separate services as shown in *Figure 4*, or the document might contain two message structures, one of which conforms to the common view.

The downside to this is that it could become a performance bottleneck, and be burdened with irrelevant data in relation to specific usage. Again it depends on the specific scenario. If you want to provide a service that returns information at the customer level, then returning the balances for all the customer's accounts in one go would reduce traffic, and simplifies the service request. However, if the requirement is purely to return the balance for a specific account, why incur the overhead of collecting and returning information on all of them from various back-end systems?

Standards Compliance

Compliance with domain standards might not be seen as a matter of principle, but as a matter of convenience. A key reason that standards help with adaptability is that they have considerable inertia and take a long time to change. So the standards provide a helpful pattern/framework for identifying certain aspects of the requirements as common and slow-changing. Standardization leads to commoditization, which may be attractive to the service consumer because it enables agility through the ability to choose from multiple providers who all conform to the same standard. Similarly it might enable a service provider to enter an existing industry by simply conforming to existing standards without the need to create new service types.

However, slavish conformance with standards might be seen as stifling agility, and an innovative or foolhardy company might well choose to deviate from industry standards in order to increase flexibility or to offer a differentiated service.
Compliance can include adoption of standards in the published interface for:

– Business semantics and schemas (rather than exposing the proprietary formats used in the underlying implementation).
– Data values such as reference data. By which we mean that participants agree to assign standard values to data where relevant. For example, agreement that airports are identified using the IATA standard (LHR for London Heathrow), or that location names conform to UN/LOCODE standards (GB LIV for Liverpool in the UK).
– Business processes, such as the sequencing of messages.

Compliance should also imply a commitment to maintain service in line with the evolution of those standards.

Standards are likely to be set at a vertical industry level by a suitable industry body, for example ACORD in the insurance industry. Some have broader applicability such as the Uniform Code Council standards for product coding. The advantages of Standards compliance include:

– Broadest compatibility between service providers and consumers.
– Existing standards based services should work immediately for new service consumers who also comply with standards.
– Standards-based services from alternative providers can be requested with the minimum, or no impact on the consumer's application; enabling dynamic switching of service provider.

Granularity

Granularity refers to the scope of functionality provided by a service. It has become best practice to recommend that Web services should be coarse grained. This is in part a reflection of the fact that the initial view of Web services was of a resource that would be deployed across the Internet with a slow and unreliable connection. As such, using a small number of coarse grained messages would reduce the number of transmissions and increase the likelihood of a message arriving and a transaction being completed whilst the connection was still available. Although the quality of connection is improving all the time, this remains good advice when delivering Web services for external use across the Internet.

However, there is now much wider internal use of Web services where the network is faster and more stable.

Table 2. Contrasting fine and coarse-grained service operations

Coarse Grained	
Benefits	**Challenges**
All data contained in single request.	Complex data. Large message size. Complexity of dealing with potentially multiple data errors in different parts of the service request.
Can reduce need to manage state as the message carries complete context.	Could lead to false sense of state. Data valid on previous submission becomes invalid on later resubmission. Need to revalidate all the data with every resubmission.
Self-described and self-contained. Carries complete context of request.	Might be geared towards a specific scenario and not reusable in others.
Fine Grained	
Benefits	**Challenges**
Small messages containing simple data.	Might need to address way state is managed between messages. Recovery of failed executions if the network is unreliable.
Individual services can be composed in other services.	Consumer will need to understand precise sequencing of service requests and overall process.
Run time flexibility. Individual services are requested only in response to the business process flow.	Complex description in terms of sequencing, pre/post conditions, etc. Individual services have no context on its own.
	Might reflect the current implementation too closely, and impacted by changes to implementation.
	Performance overheads. Increases network traffic and overheads of dealing with multiple service requests.

A higher number of fine grained services and messages might therefore be acceptable in this situation. Some of the benefits and challenges of granularity are presented in *Table 2*. There is no rule that services should all be coarse grained, or fine grained. The ideal is that they should be *right* grained for the specific usage scenario.

Granularity Varies Across Application Tiers

In an article in the previous issue of the Journal we introduced layers of service abstraction with coarse grained business services that were composed from finer grained implementation-based services (that are a straightforward translation of the existing interfaces into Web services)[1]. Web services can however be exposed at any application tier. Vendors have made it straightforward to expose Web services directly from any database, object or component. Application package vendors might expose Web services at a number of different tiers in their applications.

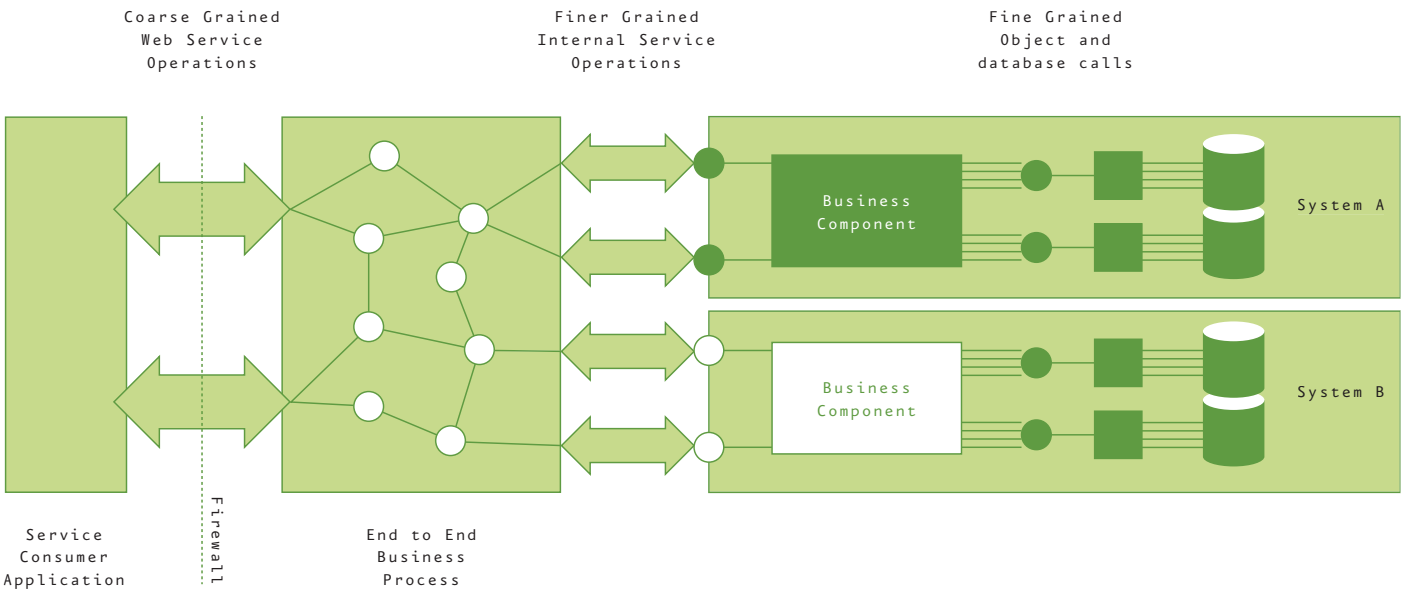
At each of the following tiers illustrated in *Figure 5* there will be different levels of granularity:

- Business Objects. Fine grained and not sufficiently abstracted away from the implementation design.
- Database. Database calls are also likely to be fine grained. Though some stored procedures might offer a coarser interface, as with components. However, either way this approach is likely to expose the internal database design which would not be desirable.
- Business Components. If you have built components to package the business objects, then their interfaces are likely to be coarser grained, and better abstracted away from the fine grained object methods. There is no guarantee of this, as it depends on the design and purpose of the component.
- Business Process Components. Likely to be coarse grained and a good match for external Web services, and a suitable level of abstraction that reflects some meaningful business service, not an internal interface.

Figure 5 illustrates that as you get closer to the requesting application the services become coarser. At the database and object tiers, then fine grained database and method calls will be implemented in native platform technologies giving scalability and performance. Decisions regarding the granularity of these are the domain of the developer.

In this example, Systems A and B are implemented in different technologies.

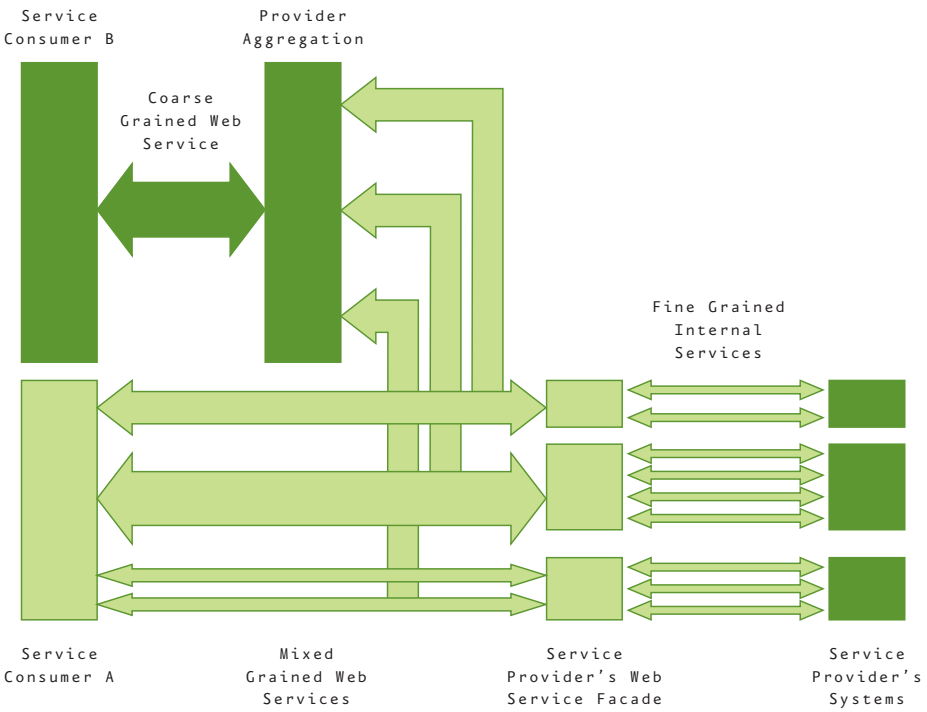
Figure 5. Service operations change granularity across application tiers



[1] *Understanding SOA*, Wilkes and Sprott. Microsoft Architects Journal, JOURNAL1, January 2004

As such Web services are used to remove technology dependencies. These Web services are also abstracted away from the implementation so that the implementation can change with minimum impact on the service. Because System A and B are distributed on the network, coarser grained services are used to reduce traffic. However, as they are still behind the firewall, the overheads of authentication and encryption are not apparent. We can refer to these as internal services. To enable reuse, these internal services are not too coarse grained so they the same service can be used in different contexts (see generalization above).

Figure 6. Delivering varying granularity of services



Provide Alternative Services. Use Aggregation and Composition

Although organizations will not want to proliferate the number of different Web services needlessly, due to development and management overheads; there is no reason to limit the provision of the service to a single Web service at a fixed granularity. For example, a business service could be exposed as a single coarse grained Web service for use by third parties, and also as a set of fine grained services for internal use, or by closer business partners. This would provide a best of both worlds solution. There are two ways that this ideal solution could be achieved. Either by simply providing a parallel Web service of the same façade, or by reusing the existing Web services and aggregating them into a new coarse grained one.

Figure 6 illustrates that services can be delivered at a number of different granularities to suit different requirements. Service consumer A might be within the same organization and have high speed reliable transport available. While, service consumer B is a third party accessing the systems across the public Internet.

Clearly there is additional cost in taking this approach. However, the following benefits can outweigh this cost:

– The cost of building a Web service is inexpensive compared to traditional interfaces.
– The availability of tools that largely automate aggregation and consolidation.
– Most importantly, it helps ensure that Web services of the *right* granularity are delivered.

A similar approach applies with generalized services. Fine grained generalized services are composed into coarser grained services that are specialized towards a specific requirement. This approach also assists with abstraction, as the coarser grained services hide the fine grained object interfaces.

Architectural Considerations

As well as adopting the above design principles we can consider some specific elements of a SOA with a view towards increasing agility.

"… there is no reason to limit the provision of the service to a single Web service at a fixed granularity."

Componentized Implementations
A widely adopted strategy to SOA is to reuse existing applications by wrapping them and exposing service interfaces. Though this is an excellent way to get off the ground quickly, it might have drawbacks with regard to longer term agility somewhere down the track.

The essence of an SOA is that the service must be implementation independent – that is, the service consumer should not have to understand the specifics of the implementation model. The problem with wrapping what already exists is that the original application is almost certainly not going to make this transparency easy at either a functional or non functional level. For example, monolithic implementations do not support the agility vision of Web services because:

– They generally don't scale well, and you cannot scale the specific part causing a bottleneck for example. You would not be able to easily scale an individual service if the implementation behind it was a monolith.
– You cannot dynamically relocate a monolithic implementation very easily.
– You cannot change or redirect one service out of many when the implementation of them all is in the same monolith – because internal dependencies create external service dependencies.

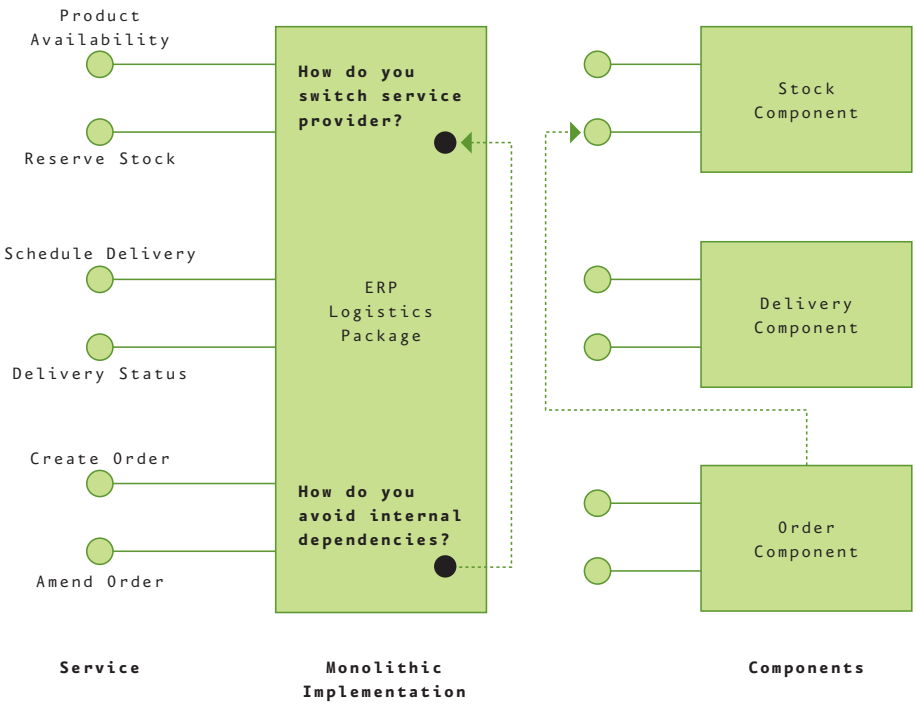
The structure of the service Implementation should not be of any consequence to the service consumer, but it still remains important to the service provider for well understood component benefits of maintainability, scalability, and so on. Moreover, a service Implementation that is not well componentized may impact the agility of service consumer who finds they are unable to use alternative services in some areas, because of the internal dependencies in the implementation as illustrated in *Figure 7*. How quickly in this example could you implement a business decision to outsource the logistics operation and use the Web services provided by the external logistics company to continue to satisfy your customers' requests for delivery information? How would you redirect an individual Web service when the internal dependencies in the ERP package still remain?

Even with a componentized implementation, developers of course still need to follow CBD best practice and ensure that dependencies are through the use of published services and interfaces, and not for example by still directly accessing the database of another component just because it is convenient.

Ideally there would be very clear alignment of the business services, the business objects and the software components that implement them at traceable level of granularity so that the response to change at any level has the minimum impact on the rest.

In the real world though, this alignment is often not present.

Figure 7. Avoid internal dependencies



"Ideally there would be very clear alignment of the business services, the business objects and the software components that implement them."

Fortunately, services help to provide an excellent transition path from existing monolithic or poorly structured implementations. By wrapping the implementation with well designed Web services you can then replace the implementation and remove internal dependencies as shown in *Figure 8*.

Finding the business justification for the final step in this transition is not always easy. Many will stop at stage 2 when they have the façade that delivers on the more immediate need to expose the business services in place. However, it should be understood that aspects of business agility, not just technical agility remain compromised by the continued use of a poorly structured implementation.

Articulation Points

In a SOA, we can introduce a number of what we term 'articulation points' around which we can introduce flexibility for both the service provider and consumer.

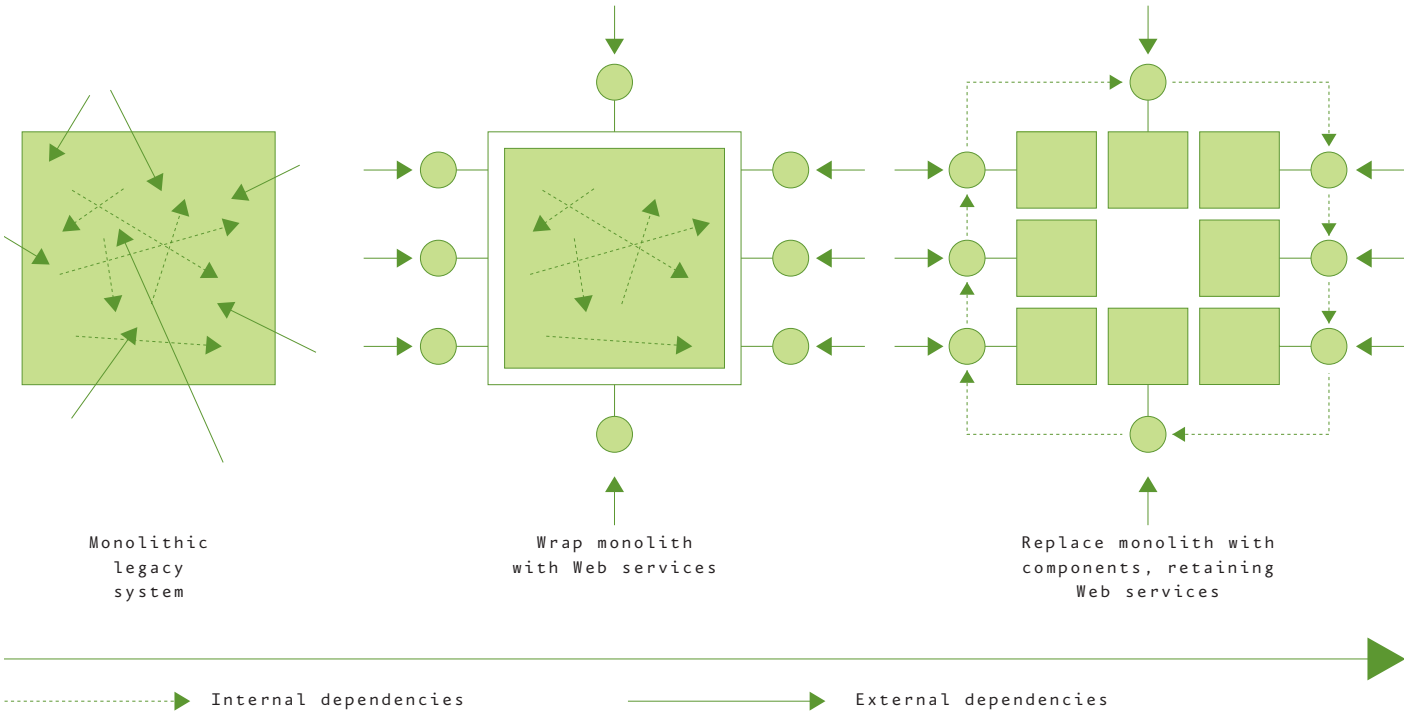
Figure 9 illustrates some numbered articulation points at which the following decisions can be made:

1. Which service should the service consumer application be using?
2. Which service provider should the request go to?
3. Which implementation-based service should receive the request?
4. Which implementation should process the request?

Agility is served best by separating these articulation points out of the applications at each endpoint so that neither the decision process nor the result is hard-wired into the code. In the process illustrated here, the application always fires off a request to the same logical service, but the actual endpoint is resolved by points 1 and 2. We can use these articulation points as described in the following list:

– Articulation point 1 might also insert further information into the request, such as identity or other non-functional information; or recompose the request so that it could be sent to an alternative provider that operates a differentiated service.
– Point 2 might also apply within the service provider, routing the request

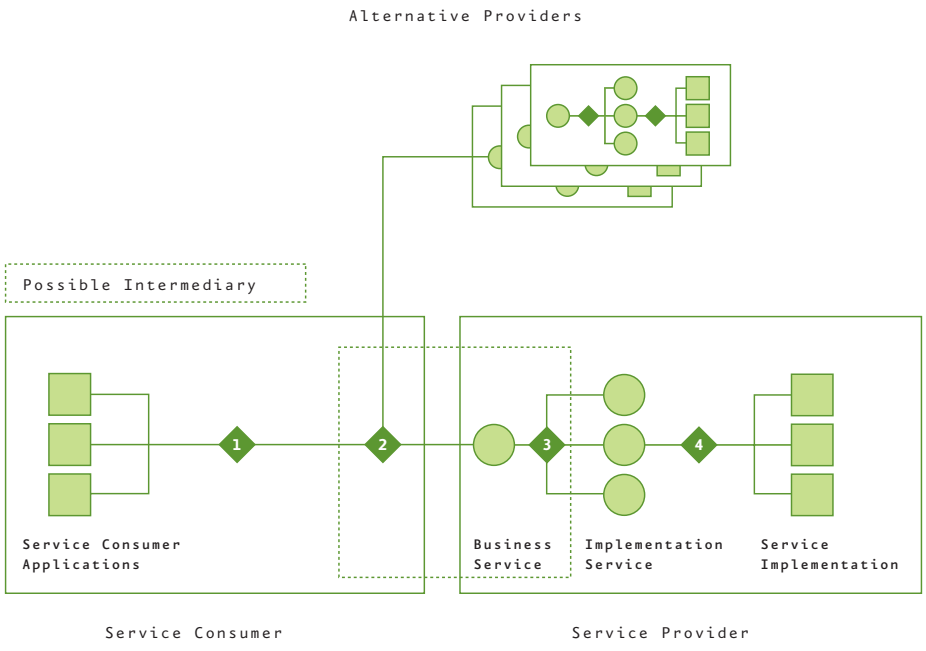
Figure 8. Using services to migrate existing applications



to different business units each with their own service.
– As well as making a routing decision at the request level, point 3 could also decompose the coarse grained request and route parts to separate finer-grained implementation based services.
– Point 4 could be routing the request based on version, or business rules such as customer type.
– One or more of these points, typically 2 and 3 could be performed by an intermediary.

There are a number of approaches to implementing each of these articulation points. While a simple script and perhaps some look-up table might suffice in some instances (and such a capability might well be built into the web server for example), often a more complex solution, or one that provides for greater flexibility and can be driven by multiple business rules, is required. Options include implementing some form of service broker which performs straightforward switching and routing, or a service façade that may include more complex functionality. For agility, the important consideration is how these points are designed into the SOA and not left to the developer's at implementation.

Figure 9. Articulation points



Service Façades

The concept of a façade should be familiar to most. Whether in the context of a building or software, the basic principle is the same; to expose a different external view by hiding that used internally by the underlying implementation. An obvious role for a Web service façade is to act as a wrapper, converting the underlying implementation technologies into Web service protocols such as SOAP and WSDL. As discussed in the previous section, the façade can be used to help migrate legacy applications. The façade can also be used to perform many other functions in a SOA such as a level of process orchestration, or Web service management.

In the context of agility, a façade can be used to deliver some of the design principles discussed so far,
– **Abstraction.** The façade not only encapsulates the implementation and provides a black box view, but can be used to further abstract the Web service away from the implementation.
– **Granularity.** The façade can be used to compose the right granularity of Web service for the consumer as illustrated earlier in *Figure 6*.
– **Versioning.** The façade can also be used to manage the introduction of new versions of the Web service (delivered as new services), and versions of the implementations.

How the façade is implemented can vary. There can be significant processing required in the façade, and the following approaches are most suitable:
– New process component that is built specific to the need of a particular service. This would be highly optimised, but not very generalized for other uses. However, it could be based on a pattern that could be quickly replicated.
– New process component that is built

at the level of a Business Service Bus, a concept we introduced in the previous issue of the Journal[2]. A BSB is a group of Services related by business domain. For added flexibility, this might be database driven in such a way that new services are easily added and composed.
– A business process, or orchestration engine, such as Microsoft BizTalk Server. We consider this a very agile solution that is capable of performing many façade tasks.

Service Brokers

A service broker provides a switching or routing mechanism. This takes the address of the incoming request and routes it instead to another. The WS-Addressing[3] protocol standard is particularly applicable here as it provides a framework for the addressing of service requests that enables their routing to be determined dynamically. As well as capability within the web server, the service broker can take many forms such as:
– A new software component built or acquired for the task and implemented in the application server
– A hardware router, some of which are now emerging with Web service routing capability
– A Web service Management (WSM) tool, several of which include brokering functionality
– Again the orchestration engine can perform this – though it may be overkill just for broking.

As can be seen in the previous discussions, there is no single solution to each articulation requirement. The products available, or those built in house can often perform more than one function and serve many articulation points.

In addition, some of the products, notably orchestration engines and WSM tools, perform other functions besides those required here. You would be unlikely to acquire one just to perform straightforward brokering functions as this would not be a very cost effective option. Acquiring these is often an organizational not a project level decision. However, when justified for their full functionality, it would make sense to put apply them to both simple and complex articulation requirements – although this must be balanced against the highly optimised and efficient processing a simpler switch can deliver in high throughput situations.

Summary

Many vendors' presentations on Web services will commence with the obligatory statement of the business problem reflecting the need for agility. However, there is a big gap between this need and the Web service technology that is subsequently presented as the solution. While Web services are an essential enabler, delivering business agility is more a consequence of thorough analysis of the business need with agility in mind and careful design of the solution that is architected for agility using the approaches outlined here.

Desirable as agility might be, delivering on the principles outlined in the report may not be straightforward as the following obstacles need to be overcome.

– Organizational culture and project orientation. In the same way that project orientation inhibited the reuse of components, it may also stifle agility. The needs of the business as a whole will not be considered, and generalized services that may be applicable elsewhere will not be delivered.
– Short term thinking. The pressure to meet immediate requirements leaves no scope to discuss future needs. This might not in itself prevent IT from trying to introduce the principles outlined here, but make it difficult to discuss and agree them with the business.
– Business Secrecy. Whilst the business may desire agility, it may not necessarily want to share with IT the business strategies that might require agile solutions.
– Ability to abstract and generalize.

[2] *Understanding SOA*, Wilkes and Sprott. Microsoft Architects Journal, JOURNAL1, January 2004

[3] http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/ html/ws-addressing.asp

The skills required are not always common in developers, particularly if it also requires sufficient familiarity with the business to be able to abstract and generalize the business concepts.

If businesses are to become more agile then they must address these issues. Business leaders cannot continue to bemoan the IT department's apparent lack of responsiveness to change when its own practices can create the above obstacles. Similarly the IT department must also play its part in designing agility into their solutions and not delegating it to the IT vendor's latest 'magic bullet'.

References

This article draws from the following CBDI Journal Reports that provide greater depth on this topic:

– *Business Adaptability and Adaptation in SOA*. Veryard, Feb 2004 http://www.cbdiforum.com/secure/interact/2004-02/business_adaptability.php
– *Best Practice Report: Component Based Service Engineering*, Veryard, Nov 2003 http://www.cbdiforum.com/secure/interact/2003-11/comp_based_srv_eng.php3
– *Design Pattern: The Web service Façade*. Wilkes, Jan 2003 http://www.cbdiforum.com/secure/interact/2003-01/facade.php3
– *Autonomic Computing*. Veryard, Jan 2003. http://www.cbdiforum.com/secure/interact/2003-01/auto.php3
– *Enterprise Service Oriented Architecture*. Sims, March 2003 (First of a five part report) http://www.cbdiforum.com/secure/interact/2003-03/foundation.php3

Lawrence Wilkes,
Technical Director and Principal Analyst, CBDI Forum
lawrence.wilkes@cbdiforum.com

Lawrence Wilkes 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.

Richard Veryard,
Associate Analyst, CBDI Forum
richard.veryard@cbdiforum.com

Richard Veryard is a consultant and writer specializing in the management of technology, and the technology of management. He has worked in the software industry for many years, with special focus on the development and deployment of products and processes, across a wide range of industries and sectors in many countries. He has written several books on modeling and management, and has published and presented widely. He is a regular contributor to CBDI, and is currently Deputy Chair of IFIP WG 8.6, an international working group on the Diffusion, Adoption and Implementation ofInformation Technology.

Service Oriented Architecture Implementation Challenges

By Easwaran G Nadhan, EDS

Ensure success by building and following a road map that incorporates enterprise-specific standards.

Introduction

You may well be considering deploying a service-oriented architecture across your enterprise. In any such deployment, there are complex challenges along the way – including ones unique to your industry and company. However, with a flexible road map for the implementation in hand you're able to act quickly to meet and overcome the challenges as they occur.

Service-oriented architectures are an important new paradigm that supports modularized implementation of solutions within a middle tier. These architectures are particularly applicable when multiple applications running on varied technologies and platforms have to communicate with each other.

However, service-oriented architectures aren't implemented overnight. Companies must first gear up and work towards the progressive construction of the components and services involved. A road map and company-specific standards are key prerequisites – ensuring a systematic implementation of such an architecture enterprise wide.

This article offers different approaches for companies to use to address various implementation-related challenges. Examples are based, when possible, on EDS' real-life experiences with our clients. The article also leverages EDS' experience in building a tool that facilitates the configuration, management and deployment of Web services within enterprises.

Architectural components

Figure 1 shows the basic components of a service-oriented architecture. The components of a service oriented architecture include:

– *Service providers*
A service provider is a component or set of components that execute a business function in a stateless fashion, accepting predefined inputs and outputs.

– *Service consumers*
A service consumer is a set of components interested in using one or more of the services provided by service providers.

– *Service repository*
A service repository contains the descriptions of the services. Service providers register their services in this repository and service consumers access the repository to discover the services being provided.

Challenges

While implementing a service-oriented architecture, a company faces up to eight key challenges. These challenges align to the steps in a typical project deployment plan:

1. **Service identification.** What is a service? What is the business functionality to be provided by a given service? What is the optimal granularity of the service?
2. **Service location.** Where should a service be located within the enterprise?
3. **Service domain definition**. How should services be grouped together into logical domains?
4. **Service packaging**. How is existing functionality within legacy mainframe systems to be re-engineered or wrapped into reusable services?
5. **Service orchestration**. How are composite services to be orchestrated?
6. **Service routing**. How are requests from service consumers to be routed to the appropriate service and/or service domain?
7. **Service governance**. How will the enterprise exercise governance processes to administer and maintain services?
8. **Service messaging standards adoption**. How will the enterprise adopt a given standard consistently?

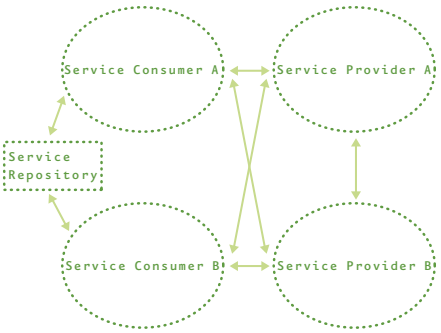
I'll discuss these challenges in detail and examine the approaches we can use to address them. Representative real-life examples are included, wherever applicable.

Service identification

Challenge

Properly identifying services and determining corresponding service providers is a critical first step in architecting a service-oriented solution. In today's world, similar business functions could very well be provided by multiple systems within the enterprise.

Figure 1. Service-oriented architectural components



"Properly identifying services and determining corresponding service providers is a critical first step in architecting a service-oriented solution."

Approaches
There are two ways to address this challenge; service rationalization and service consolidation.

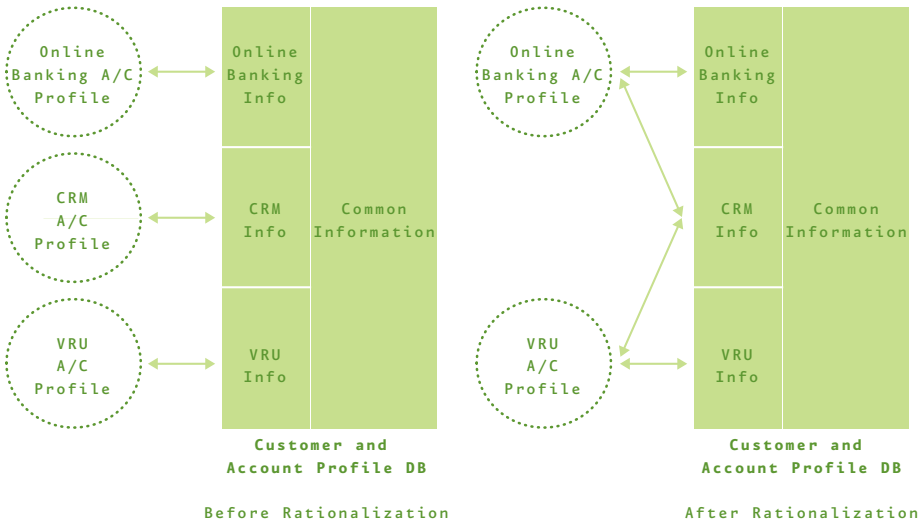
Service rationalization
Service rationalization involves a careful analysis of all the systems and applications providing the given business function. Through service rationalization, business functionality supported by the least frequently accessed systems can be implemented within those that are more frequently accessed. By streamlining systems in this way, we can enforce more consistent delivery of services.

Figure 2 provides an example of service rationalization. The information received through the Account Profile business function is required by multiple front-ending applications such as online banking, CRM and VRU applications. The customer and account repository is the system of record that supports the Account Profile business function. Depending on the nature of the front-end application invoking this business function, different subsets of the account profile are returned.

In this example, the enterprise is increasing online and VRU access for its customers while decreasing use of a CRM application that requires significant human interaction. As customers adapt rapidly to self-service channels, the percentage of access through the CRM application steadily decreases.

As part of the service rationalization process, the VRU- and online banking-based Account Profile services are augmented to implement the CRM Account Profile business function, as well. Thus, rationalization eliminates the CRM Account Profile service and the definition of two services supporting it.

Figure 2. Account profile service rationalization

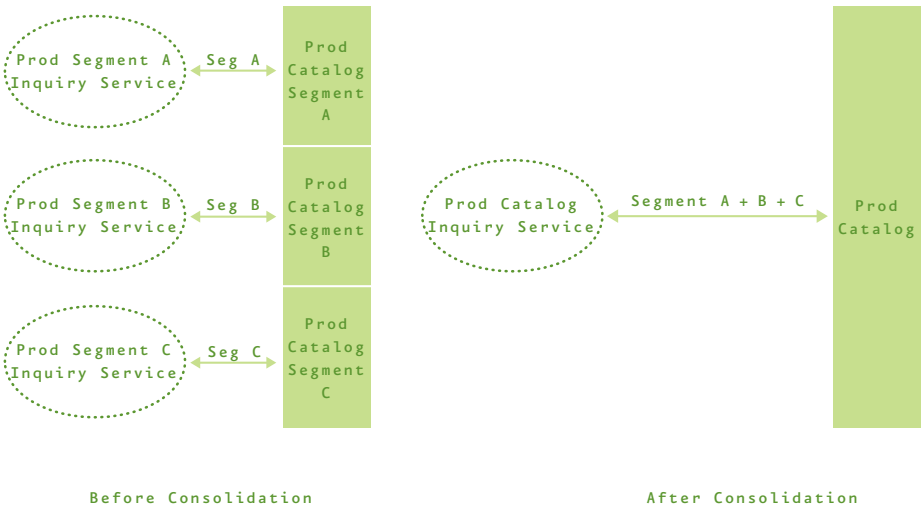


Service consolidation
Service consolidation involves the redefinition of all the service instances into a consolidated version that supports the superset of all the interfaces exposed by the individual instances. The redefined and consolidated service is provided by all the individual applications consistently.

Figure 3 illustrates a product catalog repository accessed by three separate services. These services are dedicated to retrieving predefined subsets of the information available about a product. After service consolidation, there's a single service that works with the whole product catalog. This service contains all the information segments employed by the individual services prior to consolidation. Service consumers selectively work with the portion of the catalog that's of interest to them. Service consolidation is thus an effective way to streamline multiple services supporting the same business function.

Service location
Challenge
Services usually operate on a specific set of business entities that are resident within a given system of record. This system of record is an ideal location for the service to execute. However, distributed architectural solutions can result in business data being spread across multiple applications and can generate multiple instances of the system of record for the same business entity. Data synchronization between the two systems becomes a key requirement. Where would the service be located in such scenarios?

Figure 3. Product catalog service consolidation

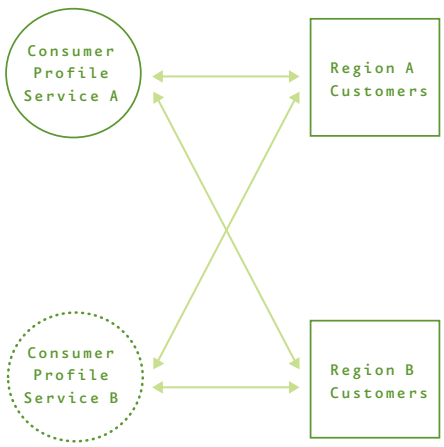


Approaches
There are three ways to solve this challenge; content-based routing, service repository-based routing, and back-end replication.

Content-based routing
This approach routes the incoming request for this service to the appropriate system of record. Such a solution supports location transparency for service consumers: The algorithm that determines where a given service is provided doesn't have to be exposed to service consumers. Both systems of record support an instance of the service and both service instances serve as logical entry points for the given request.

Figure 4 illustrates an example of content-based routing. In this example, information about customers is segregated by region. Customers belonging to a given region are stored in the repository in the data center located within that region. However, service consumers located within either region may access this information. Upon receiving the incoming request, the Customer Profile service executes a business rule that determines the specific repository where the information about the given customer is available. Then, the Customer Profile service routes the given request to the appropriate region.

Figure 4. Content-based routing



Service repository-based routing
A variation of the content-based routing approach described above, the service repository-based routing approach is shown in *Figure 5*. While the Customer Profile service executes the same business rule it does in the content-based routing approach, it leverages the information in the service repository to direct the request to the appropriate region. This approach makes it easier to change the routing logic, if necessary. Requests can be redirected to a different region simply by updating the information in the service repository – without changing the business rules within the Customer Profile service itself.

Back-end replication
This approach leverages intrinsic inter-application connection capabilities to access the information from the physical repository that contains the required information. Thus, both instances of the system of record function as a logical entry point to access the information distributed across both systems. The service can be executed on either system. The physical location of the data being operated upon is transparent to the service itself. *Figure 6* illustrates a scenario for back-end replication. The same Customer Profile service is executed on the instance of the system of record that's closest to the service. In the event that information from the other regional repository is required, the intrinsic data replication capabilities of the technology behind the data repository are employed to fetch the relevant data.

Figure 5. Service repository-based routing

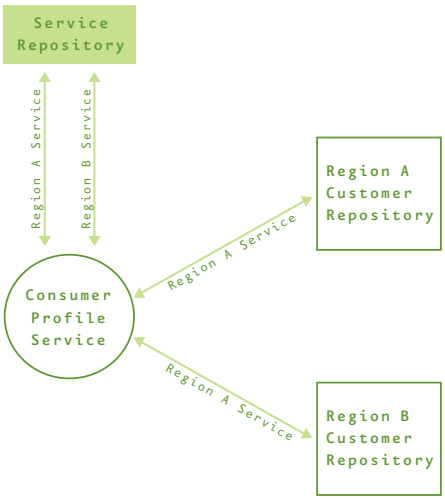
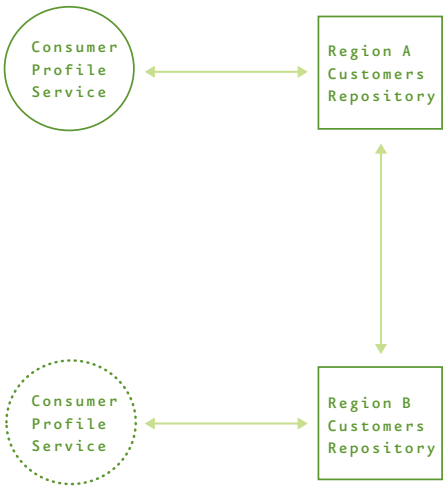


Figure 6. Back-end replication



Service domain definition

Challenge

Classifying services into logical domains simplifies the architecture by reducing the number of components to be addressed. Such groupings can be leveraged for multiple architectural reasons such as load balancing, access control, proxy simulation, and vertical or horizontal partitioning of business logic. However, it's often a serious challenge for business units and technology centers within an enterprise to come to a consensus on an appropriate definition of service domains. What would be a good logical grouping of service domains?

Approaches

We can adopt multiple approaches to define service domains. *Table 1* shows a sample distribution of the applications and platforms across different business units. This example will be used to define the salient characteristics of each approach discussed in this section.

Functional domains

Functional domains are based on the business functions being served by a set of services. The business process owners within the enterprise are best placed to define and segregate the business functions and, therefore, the service domains. Through such groupings, business process owners for a given domain can have autonomous control of the services within that domain. As long as business process owners ensure that specified services within their respective domains are provided to the rest of the enterprise, they have complete control over the architecture and implementation of the services.

In the example above, there are three functional service domains: Loans, Banking and Insurance. Services housed within these domains may have to go across multiple platforms and back-end applications to process the incoming requests specific to their domains. However, the business processes served by the services are similar within a given domain, regardless of the application or platform on which the services are executing.

1. **Loans** The Loans service domain houses services typically provided within the context of issuing and managing loans to consumers and corporate entities. This service domain includes both mortgage loans for purchasing residences as well as loans to purchase assets other than residences. Services may include loan origination, loan amortization and monthly payment calculation.
2. **Banking** The Banking service domain houses services typically associated with banking through multiple media such as the Internet, ATM, VRU and financial centers. Possible services include opening an account, retrieving an account balance and transferring funds between accounts.

Table 1. Sample application distribution

Business Unit	Primary Middle Tier Platform	Application
Home Loans	UNIX	SAP
Online Banking	Windows	Siebel
Banking Centers	UNIX	PeopleSoft
Insurance Services	Windows	SAP
Consumer Loans	Linux	Oracle
Corporate Loans	UNIX	IBM DB2

3. **Insurance** The Insurance service domain contains services that are unique to the insurance industry. Possible services include premium computation, medical history lookup and claims processing.

Technology-based domains
Functional service domains that span multiple technology platforms pose the intrinsic challenge of keeping pace with the state of each technology platform at any given time. Vendors tend to interpret industry standards in a way that favors their solution and that forces companies to depend on their architecture, hardware and/or software. Specification of service domains based on technology allows efficient and effective usage of that technology's unique capabilities.

In the example above, service domains may be classified by the UNIX, Linux and Windows platforms. Infrastructure services such as error logging, transaction monitoring and event handling are good candidates for such services. They're dependent on the platform that they're executing on and are typically independent of the business processes that drive the functional service domains.

Application-based domains
The concept of enterprise application integration came about as a way for companies to eliminate the need to replace existing systems. Today, enterprises have many multiple front-ending applications that need to integrate with the same system of record to process, package and present the same information in different ways.

Application-based service domains allow for grouping services provided on a given system. Such an approach eases the administration and maintenance of the services since the underlying system is the same for all the services within the domain.

In the example above, SAP, Siebel, PeopleSoft, IBM DB2 and Oracle are good candidates for application-based service domains. Some of the sample services that may be housed within these domains are listed below.

SAP
– Accounts payable – accounts receivable reconciliation
– Financial accounting
PeopleSoft
– Addition of employees
– Compensation updates
Oracle
– Data replication
– Role-based presentation of account information

Service packaging
Challenge
In a service-oriented architecture, an enterprise's systems must expose functionality as services. Systems built to facilitate integration can do this more easily; mainframe-based legacy systems have more difficulty. When these systems were built, they served as monolithic applications containing all the business rules and processing logic involved. Such information was distributed across multiple sets of interlinked programs.

A service-oriented architecture encourages the individual services to be self-contained – with no knowledge of the context of the other services. Mainframe programs are deeply intertwined with context-specific knowledge. How are such mainframe programs to be re-packaged into independent, self-contained services?

Approach
We can use a three-step approach to address this challenge. The approach involves defining the logical business areas within the mainframe solution, assigning program sets to these business areas, and then engineering a loosely coupled solution between the program sets. These steps are outlined in more detail below:

1. **Business area definition**
   In this step, we establish logical areas of business functionality. We can use program call maps and process flow diagrams to define these business areas. We can also define the areas by leveraging relationships between the programs on the mainframe systems. [Programs on mainframe systems tend to be linked to one another. There are common underlying business processes behind such program-to-program relationships.]

2. **Program assignment**
   Having identified business areas, we assign individual programs to a given business area. We many need to re-engineer programs that do not readily lend themselves to a specific business area so they align better with the given business area. Such a grouping of programs aligns well with the service domain concept discussed earlier.

"A service-oriented architecture encourages the individual services to be self-contained – with no knowledge of the context of the other services."

3. Loosely coupled integration
At this point, even though the programs have been assigned to the identified business areas, they're still interlinked amongst themselves. In this final step, we replace that tightly coupled relationship with a more loosely coupled approach. To do so, we redefine mainframe program interfaces so that other applications can leverage them; the programs provide the same inputs and accept the same outputs that they did originally.

This redefinition process provides an excellent opportunity to ensure that these programs serve the enterprise as a whole rather than serving single applications in isolation – the purpose for which they were originally created. This approach also gears up the existing mainframe programs toward a more service-oriented approach, positioning them as services used by service consumers external and internal to the mainframe systems.

Service orchestration

Challenge

A given service exists because there's at least one instance of a service consumer initiating the request for that service. In some scenarios, however, a service may have to invoke many other services to fulfill service consumer's original request. Simple scenarios involve a given service extending the original request to one or more services. However, complex scenarios can involve recursive invocation of multiple services and, in some extreme cases, inter-dependent invocation of multiple services – which could result in a deadlock.

Here's an example. For an airline ticket to be purchased, the following services need to be executed:
– Get Customer
– Get Schedule
– Check Availability
– Quote Fares
– Receive Payment

Building the orchestration intelligence into each service can result in the rather complex scenario illustrated in *Figure 7*.

How can such composite services be orchestrated?

Business process management approach

This approach keeps the individual service simple: The services don't have the intelligence to orchestrate the procedural invocation of all the other services required to fulfill the request. Instead, that intelligence is placed within the business process layer. Business processes are responsible for procedurally invoking each constituent service, thereby providing the composite service the service consumer originally requested. The business process becomes a specialized instance of a composite service.

Figure 7. Service orchestration challenge

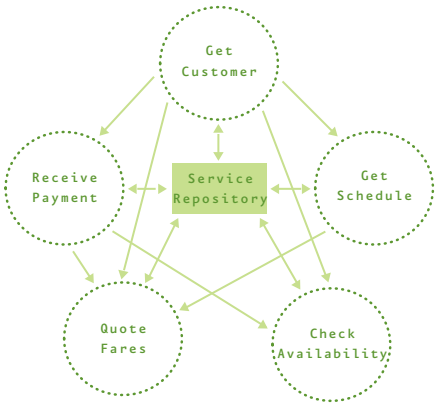


Figure 8 illustrates the Purchase Ticket business process that contains the procedural logic of the individual steps to be executed. The Purchase Ticket business process discovers the constituent services through a single access to the service repository and subsequently orchestrates the appropriate steps in sequence.

Figure 8. Business process management approach

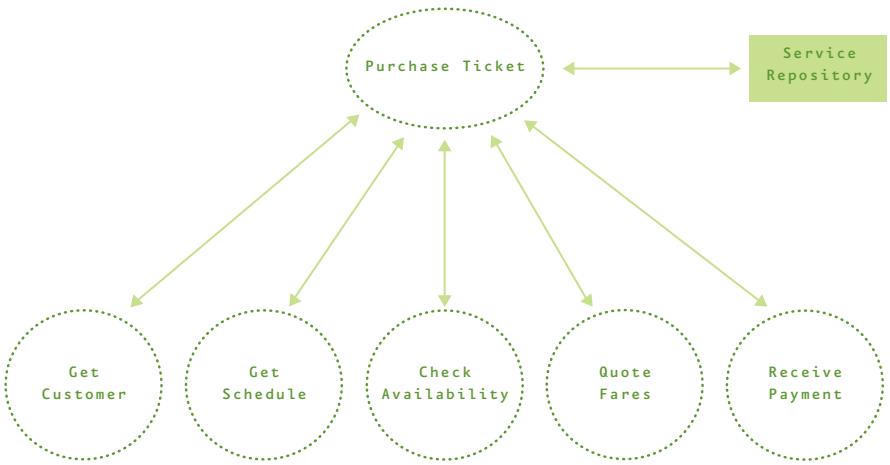
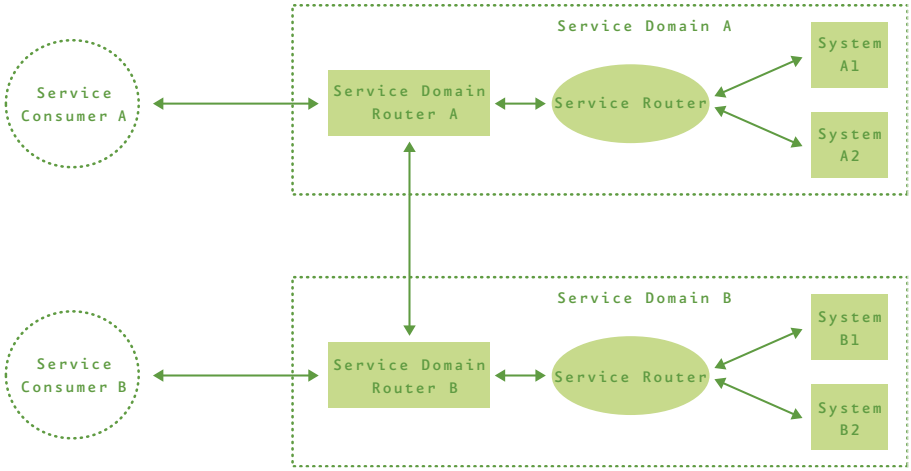


Figure 9. Service domain routers and service routers



Service routing

Challenge

Service-oriented architectures must provide location transparency to the service consumers: Service consumers have to be able to send a request for any service located in any service domain. At the same time, accessing the service repository before each invocation of a service can be a time-intensive process. How can these architectures provide location transparency while also ensuring acceptable system performance levels?

Approaches

We can solve the service-routing challenge in two ways.

Intelligent services

Using this approach, we build location information for all services into each individual service. This eliminates some of the hops required but results in an overloaded service. Depending on the frequency at which the services or their locations undergo changes, this approach can be maintenance-intensive. In addition, such an approach isn't in line with the loosely coupled architecture embraced by services. Nevertheless, it supports a high-performing solution.

Routers

The other approach is to move the routing intelligence from the individual services to a routing component. These routing components can be at two levels: service domain and service.

1. Service domain router

A service domain router has intelligence about the location of all service domains. Upon receiving a request, it determines if it can service the given request by using one of the services it supports.
If so, it processes the request.
If not, it passes the request on to the appropriate domain that can service the request.

2. Service router

A service router is used within a service domain to direct the incoming request to the appropriate service within the domain. Only those requests that can be serviced within a given service domain are passed on to the service router. The service router reduces the load of the location information on the individual services.

Service domain routers and service routers are more applicable to service domains that contain a significant number of services. When there are only a few services, intelligent services are a viable option.

Figure 9 illustrates the concept of service domain routing and service routing within a domain.

Service governance

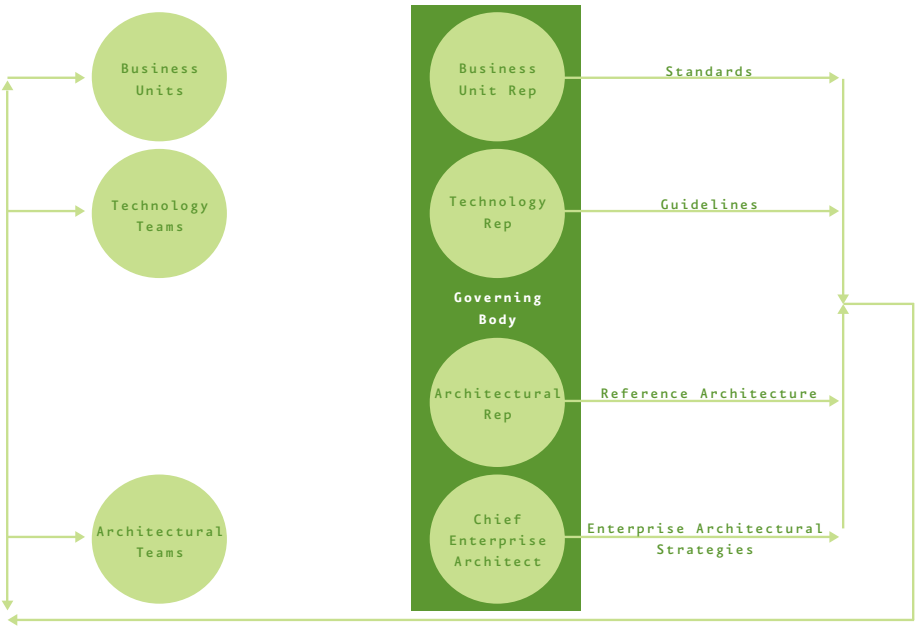
Challenge

Regardless of the way service domains are defined within an enterprise, there are various philosophical and technical approaches for creating new services and modifying existing services. Who should monitor, define and authorize the changes to the existing suite of services supported within an enterprise? Who should own the provisioning and maintenance of these services?

Approaches

An enterprise can address the challenge most effectively by establishing an internal governing body. Multiple governance models are possible. These are discussed below.

Figure 10. Central governance model



Central governance
With central governance, the governing body within the enterprise has representation from each service domain as well as from independent parties that don't have direct responsibility for any of the service domains. There must also be representation from the different business units and from subject matter experts who can speak to the key technological components of the solution. The central governing body as a whole reviews addition and deletion of services, as well as changes to existing services, before authorizing their implementation.

As shown *Figure 10* above, the central governing body is responsible for establishing and enforcing service-oriented architectural guidelines and standards across the enterprise.

The body is also responsible for communicating those standards to the business units, architectural teams and technology teams.

Distributed governance
With distributed governance, each business unit has autonomous control over how it provides the services within its own organization. Distributed governance mandates a functional service domain approach. A service architecture committee can still provide high-level guidelines and standards for implementation of services, but that committee doesn't have to authorize changes to the existing service infrastructure within a business unit. The committee suggests compliance with these guidelines but does not enforce it.

In the distributed governance model shown in *Figure 11*, business units A and B have the freedom to establish their own independent standards. Yet appropriate passive measures (architectural and procedural guidelines) are in place for the units to follow.

Service messaging standards adoption
Challenge
Messaging standards specific to vertical industries enforce standardization on a set of data elements and message formats. However, at an individual data element level, these standards are flexible enough so that enterprises can tailor them to conform within the enterprise-specific business context. As a result, different business units within the same enterprise can conform to the same standard in multiple ways. Additionally, these standards provide for the creation of custom data elements.

For example, the Interactive Financial Exchange (IFX) standard specifies multiple ways to uniquely identify a customer:

- `<CustPermId>`, which is an internal database key that uniquely identifies a customer.
- `<CustLoginId>`, which is the ID used by the customer to log in.

Both fields are unique. Enterprises adopting the IFX standard have to decide when and where to use each field. In some cases like this one, clients have decided to ignore both and create a custom field that adapts better to their enterprise system of record!

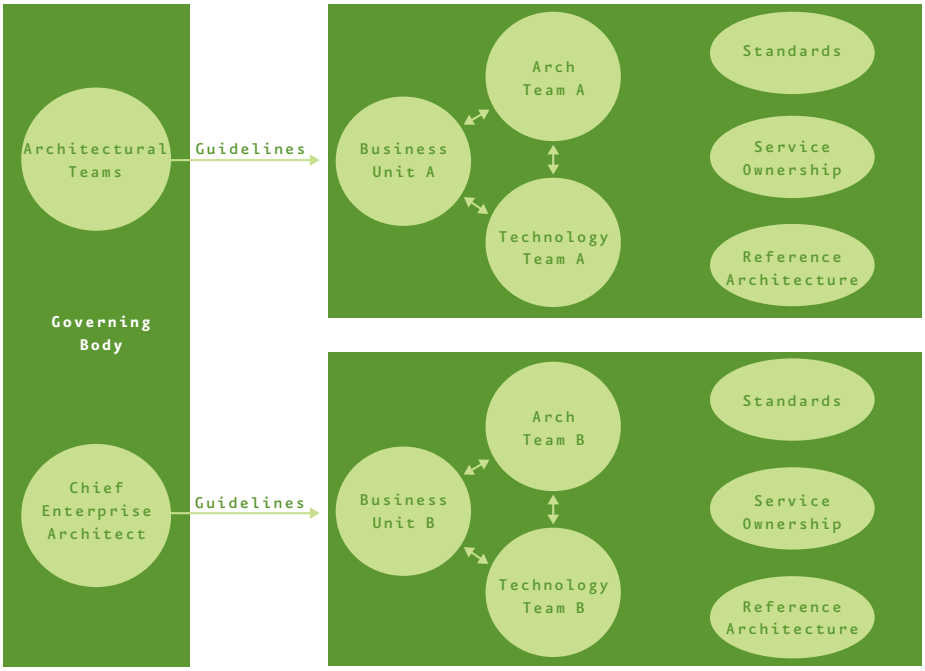
How is it possible to enforce the adoption of a single standard across the enterprise?

Metadata governance approach
Metadata repositories within the enterprise support the consistent representation of key business entities. These representations are a superset of the information distributed across multiple enterprise systems. Data dictionaries as well as logical and physical data models are key inputs into the definition and maintenance of the metadata repositories.

The metadata governance team should be a focused group within the central governance model discussed earlier. Metadata governance must be performed at the enterprise level. In other words, even if an enterprise adopts a distributed governance model for the maintenance of services in general, it must adopt a central governance model for metadata.

In the example above, the metadata for the customer would contain a single way of uniquely identifying the customer. In addition, the metadata would ensure a consistent way of representing the authentication information, including login and password. So even if a company had chosen to uniquely identify the customer by using a custom field, the field would be represented as such in the metadata repository.

Figure 11. Distributed governance model



Conclusion
Service-oriented architectures are rapidly being accepted by the IT world as a sound, modularized approach for building and deploying services across the extended enterprise. However, practical implementation of these architectures requires careful planning. Interested enterprises must first make sure that they're geared up to implement and support them in the long-term.

By developing and following an implementation road map, companies can proactively address a range of challenges that they'll encounter along the way. Each enterprise will face a unique set of challenges; corresponding approaches for solving those challenges will vary, as well. The impact of the challenges – both during and after the implementation – also depends on the context of the given enterprise.

Easwaran G Nadhan,
Principal, EDS
easwaran.nadhan@eds.com

E G Nadhan is a Principal within the Solutions Consulting division of EDS, Plano, Texas. Nadhan has designed and implemented distributed solutions in the software industry for the past 20 years. Recently, Nadhan has leveraged his experience in enterprise application integration to work with enterprises to implement service-oriented architectures. Examples in this document are based on real-life experiences encountered within EDS as well as within organizations that EDS serves.

Business Patterns for Software Engineering Use – Part 1

By Philip Teale, Microsoft Ltd and Robert Jarvis, SA Ltd

Introduction
This is the first article in a series of two. The purpose of these articles are to explore whether business patterns can be defined in structured way, and if so – whether these business patterns would be of value for those who build software systems that support the business.

Patterns
The three key parts of the pattern definition are:

– **Generic Solution** – this means that a Pattern does not define a specific solution. Rather, it identifies the 'class' of problem and how that problem might be solved with a particular approach, based on some demonstrable evidence. **Its power is derived from the fact that it is an abstraction which can be leveraged across a large number of situations.**

– **Recurring Problem** – this means that Patterns are useful when the problem is not unique, and **are most useful when the problem occurs a lot**

– **Defined Context** – this means that you have to put bounds on the generic solution because there are no universally true solutions (at any useful level). So you have to **understand the circumstances in which this generic solution is valid,** and hence how to elaborate on it to create your own specific design.

The first part explores how to define business patterns in such a way that they could be useful for software engineers. The second part explores how to develop systems from such business patterns.

In these articles the term 'pattern' is used according to the classic definition created by Christopher Alexander [1], which identifies *three key elements of a pattern.*

A Pattern describes a *generic solution* to a *recurring problem*, within a *defined* context.

Summary
What is a Business Pattern? A business pattern describes a re-usable approach to the solution of a particular business problem, usually scoped by a business process. It offers a solution based on previous success in defining solutions to the same, or similar, business problems. A business pattern may be described as an 'Architectural Template for a Business Solution'.

This paper answers the following questions:
1. Can we develop a *framework* for the creation of Business Patterns?
2. Can we define a *consistent approach* to creating Business Patterns?
3. Can we *prove that the framework and approach work?*
4. Can these Business Patterns be used to *drive either software design and implementation patterns, or actual solutions?*

In this vision and feasibility paper we assert that answer to all these questions is *yes*.

We believe this to be the case because in this article …
1. We identify the set of architectural elements needed to fully describe business patterns. We classify these and focus on the elements that describe the most stable parts of a business, which are most suitable for 'patternisation'.
2. We have defined an appropriate approach, based on tried and trusted methods.

In part 2, we provide an example from the Healthcare industry, which is based on a real engagement in the UK.

This article does not try to convince you that you should create business patterns, nor does it provide a cost/benefit case for such an activity. It describes some of the benefits to be gained from using business patterns but is not comprehensive. Our goal is to stimulate interest in this concept and sow some ideas, because we believe this will lead to better engineered software systems.

Benefit of consistency in approach
The benefit of having a framework for software engineering is well known so we do not elaborate here. However the benefit of a consistent approach is worth a short explanation.

It is the authors' opinion that we HAVE to document business patterns in a *consistent, structured* way for the following reasons:
– Such an approach allows for the gradual introduction of business patterns and business pattern-based solutions. It enables an evolution of systems that implement the patterns.

"A Pattern describes a generic solution to a recurring problem, within a defined context."

[1] *A Timeless Way of Building*, Oxford University Press, 1979

This is necessary because enterprises are usually only interested in investing in parts of their business at any one time.
– Hence we need to deliver business patterns incrementally, according to the benefit to the enterprise. Thus we need a consistent approach that allows growth in the scope of the patterns by seamlessly adding to the existing ones.
– We also believe that this approach should be used across industries, so that if an enterprise in one industry acquires another in a different industry (for example, a bank acquiring an insurance company) they have the option to integrate quite easily at the business pattern level.

The implementation interface for business patterns

We believe that the key to success is that business patterns enhance the development and maintenance of IT systems by offering a consistent interface for all business patterns. We recommend that this Interface takes the form of a definition of the components and services that will implement defined business functions on described business data.

Introduction

Models Used

We need two models – one to show how to define business patterns and one to show how to engineer systems based on business patterns, which may use other patterns in the process.

We've selected two models that we've used before, but it's important to recognise we're not saying that you have to use these models. We do feel that it is best for all if the industry uses a consistent approach to *define* business patterns; but that it could then use *many approaches to develop* the systems that implement them. The models we use have been used by Microsoft consultants both on Microsoft internal projects, and on external to create deliverables for customers and partners. These methods and techniques have been proven in the real-world.

Model 1: an enterprise architecture framework – SAM[2]

In the first part we develop a business patterns framework using SAM – the Strategic Architecture Model. SAM provides an excellent analysis approach and documentation structure for enterprise architecture and associated problem domains.

SAM uses the notions of 'spheres of interest' to represent coherent sets of facts about an enterprise and 'relationships' to associate these facts into useful groups that provide insight into the structure and operations of the enterprise. SAM has been developed by Systems Advisers Ltd,[3] a UK consultancy which has worked extensively with Microsoft in architectural areas.

In this paper we have used SAM to identify the key business pattern topics.

SAM can be regarded as a superset of the Zachman Framework for Enterprise Architecture[4] that extends this framework by providing a generic structure for architecture definition and mechanisms for organising, relating and analysing architectural information.

SAM takes an iterative approach to architecture development operating in either a top-down or bottom-up manner, or in a combination of the two, in building its deliverables. A SAM 'sphere of interest' contains all the relevant information on a particular topic, such as organisational structure or business processes, filed and organised for easy access. A sphere may be populated 'bottom-up' by gathering all relevant information at a detailed level and summarising progressively into higher and higher level groupings, or 'top-down' by defining the putative higher level groupings and decomposing these in progressively more detailed levels until an 'atomic' level is reached. In practice these approaches are usually used together to build comprehensive and resilient architectural models and most importantly to verify the integrity of the analysis.

Having built and populated a pair of spheres to a sufficient degree of detail, the members of these spheres may be linked to represent the real-life relationships that drive and sustain the operations of the enterprise. The analysis and refinement of these relationships enables business optimisation and improvement.

In the context of IT Business Patterns, SAM provides means of modelling business functions and data, and their interrelationships, to derive the components and services that support a defined business domain.

Model 2: a problem refinement model

In part 2 we use a Problem Refinement Model (PRM) to show how to work from

[2] For information on SAM – see *Enterprise Architecture – Understanding the Bigger Picture,* Bob Jarvis, a Best Practice Guideline published by the National Computing Centre, UK, May 2003 or http://www.systems-advisers.com

[3] See http://www.systems-advisers.com

[4] *Framework for Information Systems Architecture*, John A Zachman, IBM Systems Journal, Vol 28, No 3, 1987, IBM Publication G321-5298

Problem Refinement Model

A generic PRM has the following steps:

1. Problem
2. Conceptual Solution
3. Logical Services
4. Physical Services
5. Implementation

For example, we can apply the PRM to the steps in building a home.

1. **Problem:** Sheltering a family in a suitable building.

2. **Conceptual Solution:** A visualisation of the property in a model of its location with its properties described in such a way that the prospective owner can relate to them.

3. **Logical Services:** a further refinement of the conceptual solution in which the standard requirements; like rooms, windows, and roof types, are selected for the building.

4. **Physical Services:** a refinement that focuses on the infrastructural needs; such as foundations, plumbing, insulation and so on.

5. **Implementation:** the final refinement brings together the logical and physical services to shown a blueprint of how this particular house will be constructed. This is the end of this architect/designers work and the design is handed to a builder. This is analogous to the roles of the IT Architect and the IT Project Designer.

a business problem through to a technology implementation. The generic Problem Refinement Model can be used to iteratively refine any problem from its initial definition to its final resolution. Our experience has shown such an approach to be very useful for solution architects and designers and so the PRM we show has been customised to their problem space. We'll show how to use that PRM to evolve the business problems (which in our case are expressed as business patterns), into IT solutions.

The PRM we use also recognises the frequent need to separate the enterprise architect's perspective from the project designer's perspective, and uses different techniques for communication with these audiences. For each audience it describes different refinement techniques to use within a Five-layer view, and the transforms between the layers. The five layers are shown in *Figure 1*:

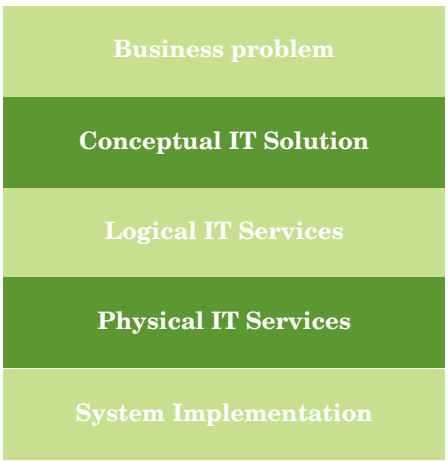
The reason for having five layers is that it seems in practice to provide a comfortable number of refinement steps:

1. The Business Problem (as represented by a pattern) is very specific to the business functions being performed and the data required.
2. The Conceptual Solution elements might be reusable by other business patterns.
3. The Logical Services are reusable across many solutions.
4. The Physical Services are oriented to nodes and function/data placement, which is vendor-product independent.
5. Vendor-specific issues are isolated to the Implementation layers. So for example, here we can provide very detailed Microsoft product usage guidance, within the overall context.

One great value of using a refinement model like this is that you can maintain traceability through the evolution of the system. So you can look at an implementation detail and trace back through the layers to understand clearly what each of the drivers for that implementation was, right up to the business problem level.

When applying the model for enterprise architects, we typically use documentation techniques that are suitable for long-term guidance of a programme of implementation projects. When using it for project designers, we use techniques like UML – commonly used 'languages' for project analysis and design. (The distinction of programme versus project need is the important point for the refinement technique used; not the role labels.)

Figure 1. A Five Layer View of Problem Refinement



"We'll show how to use that PRM to evolve the business problems (which in our case are expressed as business patterns), into IT solutions."

How to recognise & document Business Patterns

Using SAM's Spheres to Recognise the Real World

We address the question by considering a set of SAM spheres. These represent a superset of the Zachman framework.

Our experience of building real Enterprise Architectures shows that the spheres shown in *Figure 2* typically represent the important areas of interest.

These spheres of interest need some explanation. In particular it should be stressed that these definitions may vary from enterprise to enterprise. What is important is that the concepts are recognised and incorporated into the overall architectural design.

Within each sphere we store information about the particular topic – structured for easy maintenance and retrieval. Usually this will take the form of one or more hierarchical structures, the analogy of the filing cabinet comes to mind. Each item of information is known as a 'member'. A member is therefore a discrete piece of information belonging to a sphere of interest.
A sphere about 'Locations' might have the members 'Head Office', 'London Sales Office', 'Birmingham Plant', and so on. Further examples of members include:

– A specific organisational unit, for example 'sales department' within the sphere 'organisation'; or
– A specific business process, for example 'Accept Order' within the sphere 'Business Processes' or
– A particular data entity, for example 'Customer' within the sphere 'Data'.

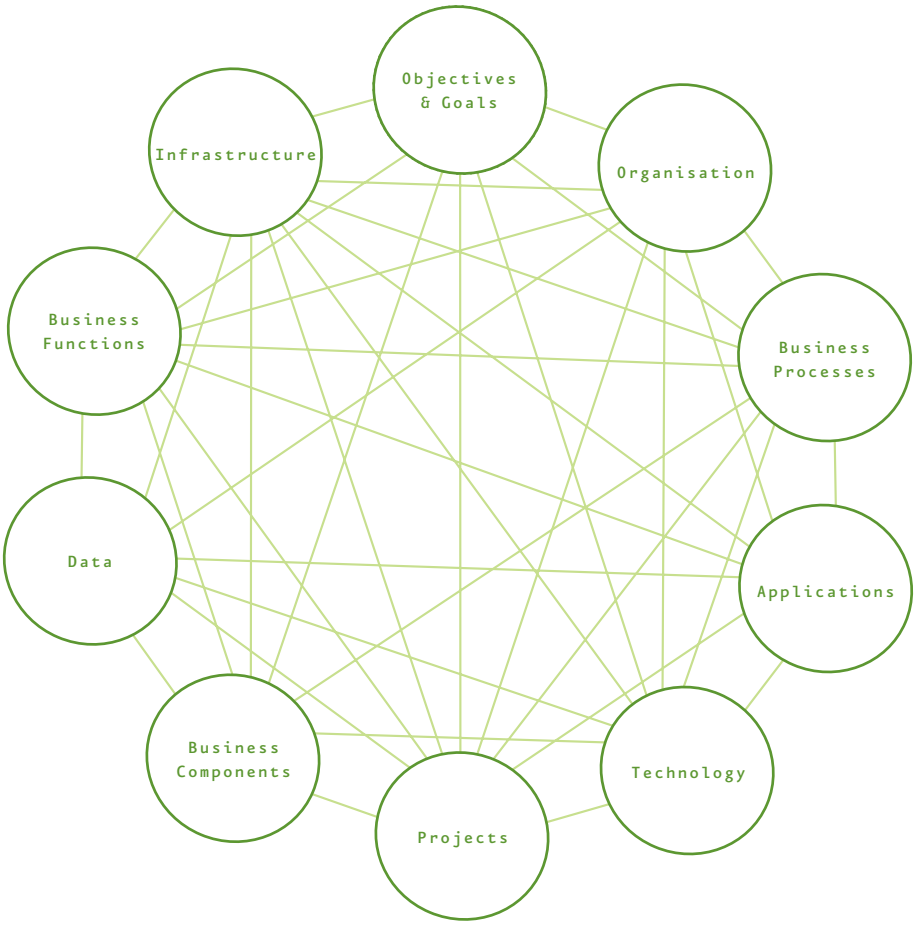
Definition of the relevant spheres

Objectives and Goals are the strategic and tactical aims of the enterprise in fulfilling its mission. They may be high-level such as 'Improve Customer Service' or quite focussed such as 'Reduce call centre waiting time to less than 30 seconds'. Objectives and goals impact business processes and are assigned to organisational units for their achievement.

Organisation is concerned with the organisational structure of the enterprise – groups, departments, divisions – and the interrelationship of these organisational units.

Infrastructure is concerned with the fixed assets of the enterprise – locations, buildings, equipment including IT equipment, networks, transportation, etc. and their interrelationships.

Figure 2. Typical Spheres of Interest



Business Processes are defined here as the procedures and activities carried out by the enterprise. Business Processes are usually expressed as a sequence of work activities carried out by various organisational units working in a co-ordinated way. Examples might be 'Process Customer Orders', 'Recruit Staff', or 'Prepare Shipping Documentation'.

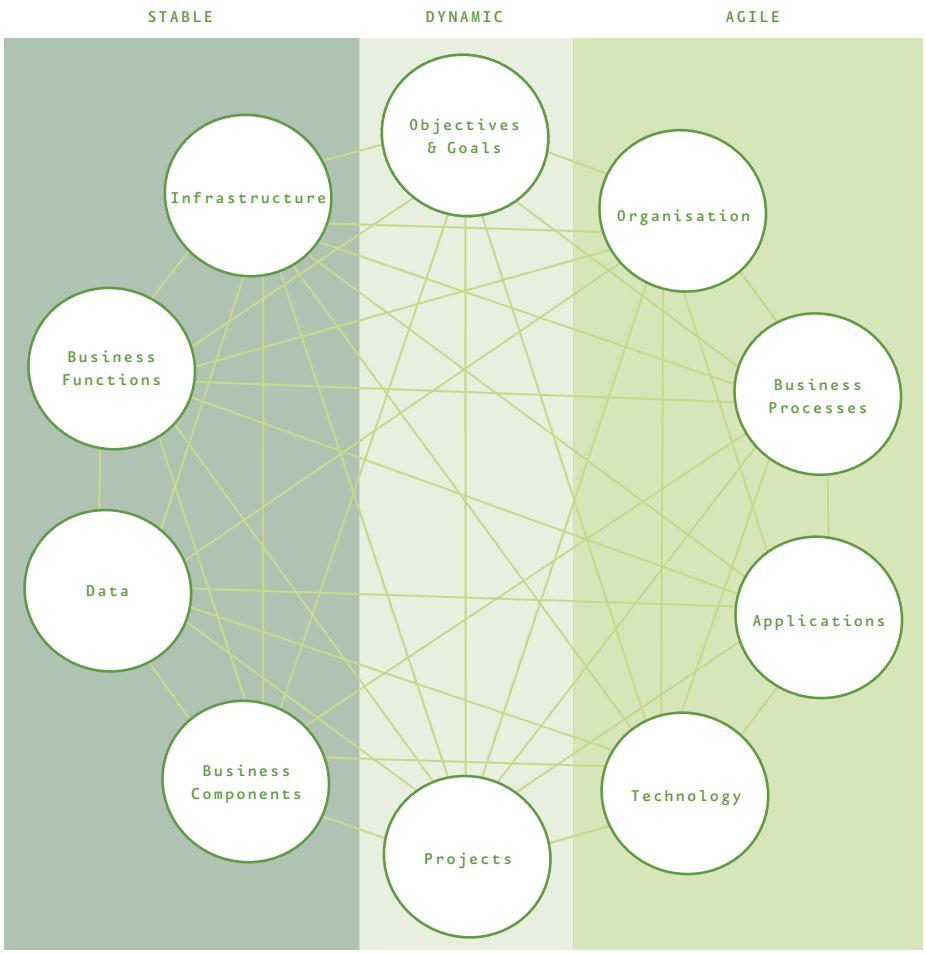
Business Functions are 'the things an enterprise does' like Marketing, Selling, Product Design, Manufacturing, Financial Management, Personnel Management, and so on. These should not be confused with 'departments' that might do these things. A function might be carried out by many departments or organisational units. Functions can typically be represented in a non-redundant hierarchy A functional decomposition is constructed on the principles of loose coupling and tight cohesion, principles of good modularisation that will be familiar to software engineers.

Data is the fundamental pieces of information created and used by the enterprise. Typically these pieces are expressed at the level of a data entity such as 'Employee' or 'Product' or 'Customer'.

Business Components are encapsulations of business function and data. A business function creates, reads, updates and deletes data. Grouping together all the functions that create and update the same data entities, using a technique such as commutative clustering, defines non-redundant 'building blocks' – components – that may be used to construct systems or applications that in turn support particular business processes. Components are also important artefacts in modern systems development. By encapsulating functionality and data, software re-use and replace-ability become practical. Further, components offer *'services'* that may be used in conjunction with the services offered by other components to instantiate a Service Oriented Architecture. Services exposed using Internet technology are called 'web services' – an important aspect of Microsoft's .NET technology.

Applications are the enterprise's inventory of computer and other systems. These would include all operational systems (the 'as-is'), those under development and those planned for the future (the 'to-be'). They may be component-based or have been built using older methods of construction.

Figure 3. Stable, Agile and Dynamic Spheres of Interest



Technology describes the hardware, software and communications environments and facilities used to construct and operate applications. *Projects* are the controlled pieces of work needed to realise an application or set of applications. Projects are prioritised in alignment with objectives and goals.

Using SAM's Spheres for business modelling

We believe that the above set of spheres and the relationships between them can be used to define a sufficient business model. So is this how we can describe business patterns? Do we do this by documenting all these spheres and all relationships?

In general terms the answer is 'Yes', but this would be is far more than is strictly necessary, or indeed desirable, because different spheres are subject to different degrees of stability: ranging from highly stable to quite dynamic. We want to base our business patterns of the stable elements of the enterprise and from this foundation create flexible, agile solutions.

We postulate that there are really three categories of sphere in the set as shown in *Figure 3*.

Set 1 = Stable

These spheres describe the stable elements of the business and represent the fundamental structures – business function, data, business components and infrastructure – that must be present in order to operate in the defined business domain.

Set 2 = Agile

The agile spheres describe the things a business does, or can do, to clearly differentiate itself from its competitors. How it does this will determine whether it is an agile business or not. The agile spheres – organisation, business process, applications and technology – are things an enterprise can change reasonably quickly, even continuously, in response to market and economic conditions.

Set 3 = Dynamic

The dynamic spheres are about business direction, work programmes and change management, basically 'what it is all about'. They describe the effort needed to move towards a set of business objectives and goals by means of a set of related projects.

Table 1 answers the questions:

– Which of these sets of spheres can be expressed as business patterns?
– Which are useful for software engineering?

Set of Spheres	Can these be expressed as business patterns?	Are they useful for software engineering?
Stable	Yes, and these would be very useful for typical business consultancies and integrators AND for software companies (ISVs) who can demonstrate how their solutions can rapidly provide rich, stable, maintainable functionality.	Yes, it's exactly what we need to do. **Hence it is what we are doing, right here!**
Agile	Yes, and again these would be very useful to typical business consultancies and, to a lesser degree, integrators who can use them to effect change in a client's business. However, there will a multitude of different patterns since the contexts and driving forces will be very volatile and numerous.	Not directly.
Dynamic	Yes, and these patterns would be very useful to typical business consultancies and system integrators who specialise in programme and change management. They can use these patterns to configure projects into programmes based upon the proven patterns from previously successful approaches.	Not directly.

"… different spheres are subject to different degrees of stability: ranging from highly stable to quite dynamic. We want to base our business patterns of the stable elements of the enterprise and from this foundation create flexible, agile solutions."

Conclusion

We conclude that a Business Pattern will describe:

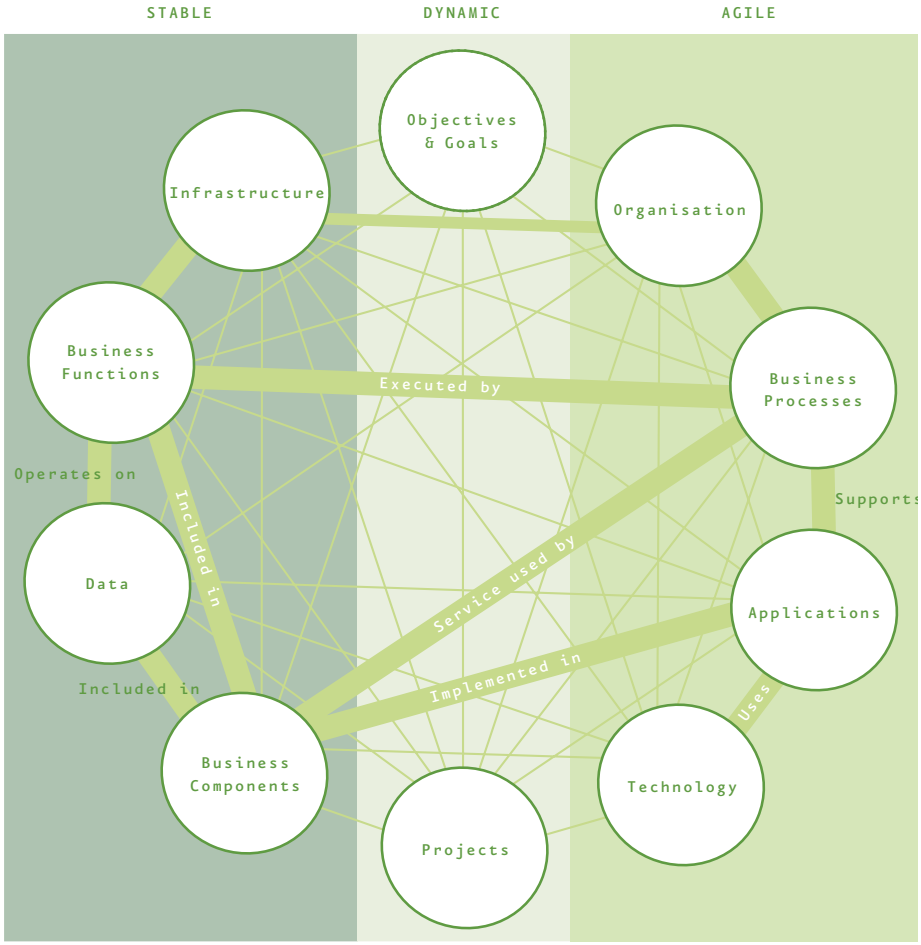
– The *Business Functions* being supported.
– The *Data* that is required to support the described functions.
– The *Business Components* that are the IT representations of the data and functions the business needs.
– Optionally, the *Infrastructure* needed to support the functions, data and components. This is necessary in highly distributed enterprises or those made up of divisions or units with diverse technical or operational environments.

In addition, the business pattern will describe the key *relationships* between these dimensions. All SAM spheres have relationships to all other spheres. However, we need to focus on certain core relation-ships that fundamentally shape the business pattern.

The core relationships are defined thus:
The Stable Relationships:
– Business Functions perform actions on Business Data (Typically create, read, update and delete)
– Business Functions are included in Business Components
– Data are included in Business Components

The stable spheres link to the agile spheres as follows:
The Linking Relationships
(Stable > Agile):
– Business Functions are executed by Business Processes
– Business Processes use the services of Business Components
– Business Components are implemented in Applications

Figure 4. Minimum Essential Model for Business Pattern Definition



The following relationships in the agile space are useful:
The Agile Relationships
– Business Processes are supported by Applications
– Applications operate using Technology
This is represented graphically in *Figure 4*.

In addition, there may be relationships involving Infrastructure and Organisation (in grey) depending upon the nature of the enterprise and the business domain being considered.

However we would issue a word of caution. Although it is far from necessary to populate all spheres before useful results appear, it is necessary to achieve a critical mass of related, stable structures before any significant decisions regarding the scope and boundaries of business patterns can be made.

Solution versus Pattern
Business patterns are defined using the stable spheres and relationships only. The connections from the business patterns to potential solutions are provided by the linking relationships. Further agility is achieved by improving and optimising the agile relationships.

In a full *Solution* Development, we would address at least the linked Stable and Agile spheres *and* the shown relationships.

Thus for a Business Pattern we require that only the three stable spheres and the relationships between them are documented. This gives the most solid base for flexibility in implementations of the patterns. The three stable spheres required are highlighted in *Figure 4*. The greyed relationships and spheres represent the linking relationships from stable to agile i.e. towards a particular solution. If a business pattern exists for the subject, then the solution builds on the pattern.

Part 2
In JOURNAL3, we will show a way of developing Business Patterns as represented by Business Functions, Data and Business Components. We will also show how these can be used to engineer software systems.

Disclaimer:
The opinions expressed in this paper are those of the authors. These are not necessarily endorsed by their companies and there is no implication that any of these ideas or concepts will be delivered as offerings or products by those companies.

Philip Teale,
Partner Strategy Consultant,
Microsoft Ltd
pteale@microsoft.com
Philip Teale is a Partner Strategy Consultant working for Enterprise & Partner Group in Microsoft UK. Previously, he worked for the Microsoft Prescriptive Architecture Group in Redmond, and for Microsoft Consulting Services before that. He has 29 years of Enterprise IT experience of which four years have been with Microsoft and 16 with IBM, in both field and software development roles. At the time he left IBM he was working in for the IBM Software Strategy group, in Somers, New York. He has also worked for Standard Life in Edinburgh, TRW in Southern California and Bank of America in San Francisco. His international experience includes nine years working in the USA, three years in Canada and seventeen years in the UK. Phil's background is in architecting, designing and building large complex distributed commercial systems, with specialisation in the Finance industry. His most recent contribution to industry thought-leadership was to drive Microsoft in the creation of patterns for enterprise systems development.

Robert Jarvis, Director, SA Ltd
v-rjarvi@microsoft.com
Robert Jarvis is a Director of Systems Advisers Limited, a UK consultancy specialising in the development of Strategic Systems Architectures for major international enterprises. He is also an Associate Architectural Consultant with Microsoft Ltd. Bob has over 30 years experience as an International Systems Consultant and Architect advising business and governmental organisations in the UK, Continental Europe and the Americas. He is the author of *'Enterprise Architecture – Understanding the Bigger Picture'*, a Best Practice Guideline published by the UK's National Computing Centre in 2003.

Messaging Patterns in Service Oriented Architecture – Part 1

By Soumen Chatterjee, CGE&Y

Introduction:
Process Configuration and Flexibility Trends

The need for process flexibility is not a new trend. The trend has been evident for the last two decades. The Internet, Web, and mobile computing came along and enabled a global productivity boom, resulting in technology innovations that are constantly laying the foundation for renovating industrial-age processes.

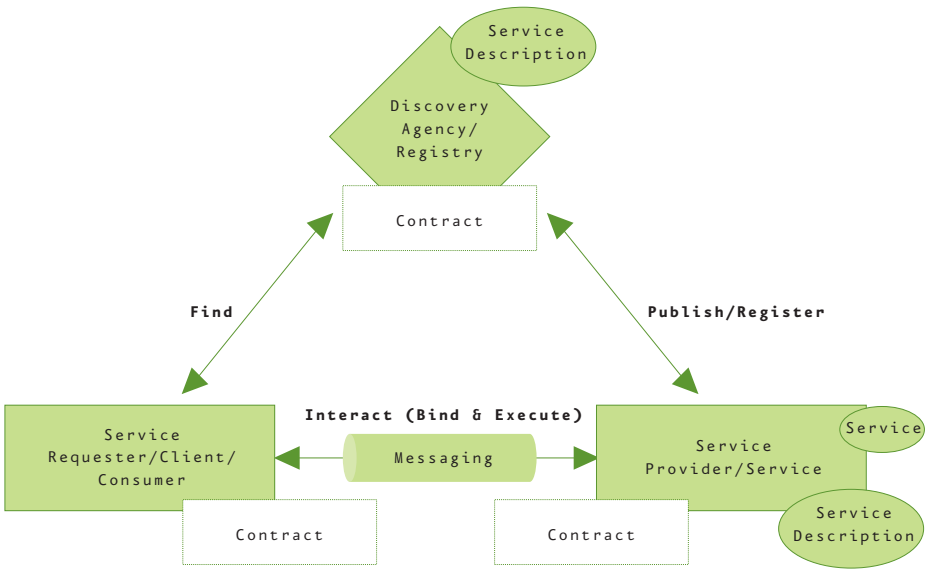
These solutions are built primarily on proprietary or system-based messaging platforms aimed at providing a platform for integration and communication between various business components. The typical method for accessing these systems is through a wide assortment of pre-built adapters that provide a bi-directional connectivity to many types of application processes. Rather than explicitly declaring how systems will interact through low level protocols and object oriented architectures, Service Oriented Architecture (SOA) makes it possible to provide an abstract interface through which processes or services can interact. It can be imagined as an interconnected process based enterprise that exposes a set of loosely coupled, coarse-grained services.

What is Service Oriented Architecture?

SOA is the aggregation of components that satisfy a business need. It comprises components, services, and processes. Components are binaries that have a defined interface (usually only one), and a service is a grouping of components (executable programs) to get the job done. This higher level of application development provides a strategic advantage, facilitating more focus on the business requirement.

SOA isn't a new approach to software design; some of the notions behind SOA have been around for years.

Figure 1. SOA Explained



Jess Thompson, a research director at Gartner, argues that the underlying concepts date back to the early 1970s, when researchers started drawing boundaries around software and providing access to that software only through well-defined interfaces.

A service is generally implemented as a coarse-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model.

The most important aspect of SOA is that it separates the service's implementation from its interface. Service consumers view a service simply as a communication endpoint supporting a particular request format or contract. How service executes service requested by consumers is irrelevant; the only mandatory requirement is that the service sends the response back to the consumer in the agreed format, specified in contract.

SOA Entities

SOA consists of various entities configured together to support the *find, bind, and execute* paradigm as shown in *Figure 1*.

Service Consumer

The service consumer is an application, service, or some other type of software module that requires a service. It is the entity that initiates the locating of the service in the service registry, binding to the service over a transport, and executing the service function. The service consumer executes the service by sending it a request formatted according to the contract.

"A service is generally implemented as a coarse-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model."

Service Provider
The service provider is the network-addressable entity that accepts and executes requests from consumers. It can be a mainframe system, a component, or some other type of software system that executes the service request. The service provider publishes its contract in the service registry for access by service consumers.

Service Registry
A service registry is a network-based directory that contains available services. It is an entity that accepts and stores contracts from service providers and provides those contracts to interested service consumers.

Service Contract
A contract is a specification of the way a consumer of a service will interact with the service provider. It specifies the format of the request and response from the service. A service contract may require a set of preconditions and post conditions. The preconditions and post conditions specify the state that the service must be in to execute a particular function. The contract may also specify quality of service (QoS) levels, specifications for the nonfunctional aspects of the service.

Service Lease
The lease (the time for which the state may be maintained), which the service registry grants the service consumer, is necessary for services to maintain state information about the binding between the consumer and provider. It enforces loose coupling between the service consumer and the service provider, by limiting the amount of time consumers and providers may be bound. Without a lease, a consumer could bind to a service forever and never rebind to its contract again.

Discoverability and Dynamic Binding: Messaging in SOA
SOA supports the concept of dynamic service discovery. The service consumer queries the service registry for a service, and the service registry returns a list of all service providers that support the requested service. The consumer selects the cost-effective service provider from the list, and binds to the provider using a pointer from the service registry entry.

The consumer formats a request message based on the contract specifications, and binds the message to a communications channel that the service supports. The service provider executes the service and returns a message that conforms to the message definition in service contract.

The only dependency between provider and consumer is the contract, which the third-party service registry provides. The dependency is a runtime dependency and not a compile-time dependency. All the information the consumer needs about the service is obtained and used at runtime. The service interfaces are discovered dynamically, and messages are constructed dynamically. The service consumer does not know the format of the request message or response message or the location of the service until the service is actually needed.

The ability to transform messages has the benefit of allowing applications to be much more decoupled from each other. Messaging underpins SOA; we don't have SOA without messaging.

Messaging Patterns catalogue within SOA context
Messaging Patterns exist at different levels of abstraction with the SOA. Some patterns are used to represent the message itself, or attributes of a messaging transport system. Others are used to represent creation of message content or change the information content of a message. Patterns are also used to discuss complex mechanisms to direct messages. SOA messaging patterns can be divided into the following categories:

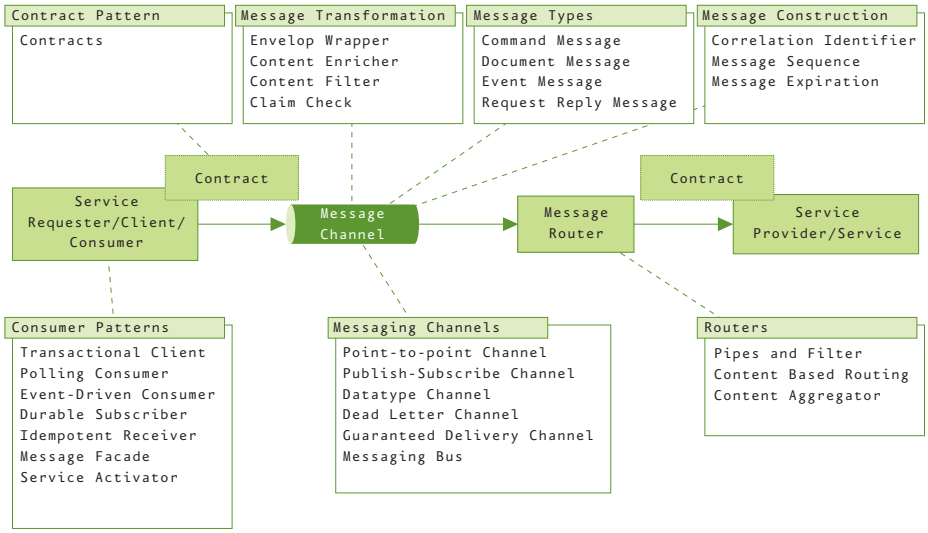
– *Message Type Patterns:* Describe different varieties of messages that can be used in SOA.
– *Message Channel Patterns:* Describe the fundamental attributes of a messaging transport system.
– *Routing Patterns:* Describe mechanisms to direct messages between Service Provider and Service Consumer.
– *Service Consumer Patterns:* Describe the behavior of messaging system clients.
– *Contract Patterns:* Illustrates the behavioral specification to maintain a smooth communication between Service Provider and Consumer.

– *Message Construction Patterns:* Describes the creation of message content that travel across the messaging system.
– *Transformation Patterns:* Change the information content of a message within the enterprise level messaging.

These patterns are shown in *Figure 2*.

"Messaging underpins SOA;
we don't have SOA without messaging."

Figure 2. Messaging Patterns Catalogue within SOA Context



Message Type Patterns

The message itself is simply some sort of data structure – such as a string, a byte array, a record, or an object. It can be interpreted simply as data, as the description of a command to be invoked on the receiver, or as the description of an event that occurred in the sender. Sender can send a *Command Message,* specifying a function or method on the receiver that the sender wishes to invoke. It can send a *Document Message,* enabling the sender to transmit one of its data structures to the receiver. Or it can send an *Event Message,* notifying the receiver of a change in the sender.

The following message type patterns can commonly be used in SOA.

Command Message

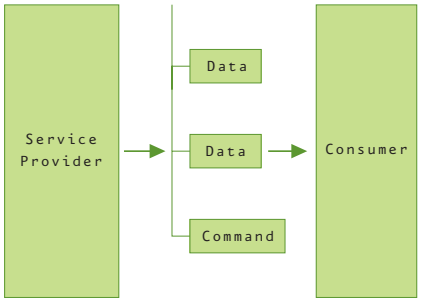
Problem:
How to invoke a procedure in another application?

Solution:
Use a command message to reliably invoke a procedure in another application as shown in *Figure 3*.

Interactions:
A *command message* controls another application, or a series of other applications, by sending a specially formatted message to that system. A command message includes intelligent instructions to perform a specific action, either via headers and attributes, or as part of the message payload. The recipient performs the appropriate action when the message is received. Command messages are closely related to the Command pattern [9].

A *command message* is simply a regular message that happens to contain a command. A Simple Object Access Protocol (SOAP) request is a command message.

Figure 3. Command Message



Command messages are usually sent on a *point-to-point* channel so that each command will only be consumed and invoked once.

Document Message

Problem:
How can you transfer data between services?

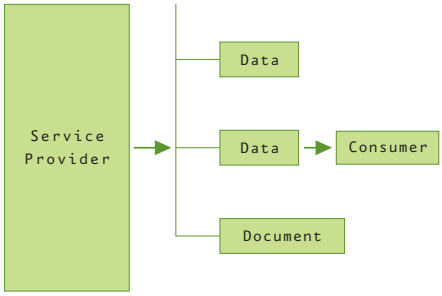
Solutions:
Use a *document message* to reliably transfer a data structure between applications. See *Figure 4*.

Interactions:
A *document message* is just a single unit of information, a single object or data structure that may decompose into smaller units. The important part of a *document message* is its content; the document. This content is retrieved by un-marshalling/or de-serializing data.

Document messages are usually sent using a point-to-point channel. In request-reply scenarios, the reply is usually a *document message* where the result value is the document.

A *document message* can be any kind of message in the messaging system. A Simple Object Access Protocol (SOAP) reply message is a document message.

Figure 4. Document Message



Event Message

Problem:

Several applications would like to use event-notification to coordinate their actions, and would like to use messaging to communicate those events. How can messaging be used to transmit events from one service to another?

Solutions:

Use an *event message* for reliable, asynchronous event notification between applications. See *Figure 5*.

Interactions:

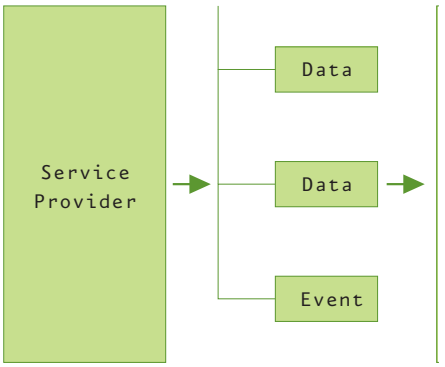
An event message extends the Observer model to a set of distributed applications. Event messages can be sent from one service to another to provide notification of lifecycle events within a service-oriented enterprise, or to announce the status of particular activities. Applications for this pattern include enterprise monitoring and centralized logging.

An important characteristic of event messages is that they do not require a reply.

An *event message* can be any kind of message in the messaging system. An event can be an object or data such as an XML document.

'If a message says that the Stock price for certain symbol has changed, that's an event. If the message provided information about the symbol, including its new price, that's a document.'

Figure 5. Event Message



Request-Reply Message

Problem:

Messages travel into a message channel in one direction, from the sender to the receiver. This asynchronous transmission makes the delivery more reliable and decouples the sender from the receiver. The problem is that communication between components often needs to be two-way. When one component notifies another of a change, it may want to receive an acknowledgement.

How can messaging be two-way?

Solutions:

Send a pair of *request-reply* messages, each on its own channel. See *Figure 6*.

Interactions:

Request-Reply has two participants:

– ***Requester (Service Consumer)*** – Sends a request message and waits for a reply message.

– ***Replier (Service Provider)*** – Receives the request message and responds with a reply message.

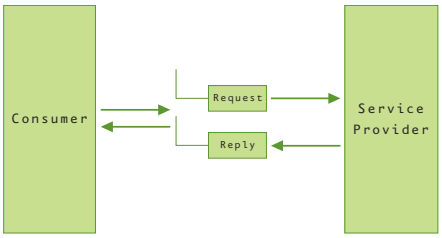
The request channel can be a *point-to-point channel* or a *publish-subscribe channel*. The difference is whether the request should be broadcast to all interested parties or should only be processed by a single consumer. The reply channel, on the other hand, is almost always point-to-point, because it usually makes no sense to broadcast replies.

The request is like a method call. As such, the reply is one of three possibilities:

– *Void*
– *Result value*
– *Exception*

The request should contain a return address to tell the replier where to send the reply. The reply should contain a correlation identifier that specifies which request this reply is for.

Figure 6. Request Reply Message



Messaging Channel Patterns

Channels, also known as queues, are logical pathways to transport messages. A channel behaves like a collection or array of messages, but one that is magically shared across multiple computers and can be used concurrently by multiple applications.

A service provider is a program that sends a message by writing the message to a channel. A consumer receives a message from a channel. There are different kinds of messaging channels available.

Point-to-Point Channel

Problem:

The sender dispatches a message to a messaging system, which is responsible for relaying the message to a particular recipient. The messaging system might proactively deliver the message (by contacting the recipient directly), or hold the message until the recipient connects to retrieve it. How can you ensure that exactly one consumer will receive the message?

Solution:

Send the message on a *point-to-point channel*, which ensures that only one receiver will receive a particular message. See *Figure 7*.

Interactions:

A *point-to-point channel* ensures that only one consumer consumes any given message. If the channel has multiple receivers, only one of them can successfully consume a particular message. If multiple receivers try to consume a single message, the channel ensures that only one of them succeeds, so the receivers do not have to coordinate with each other. The channel can still have multiple consumers to consume multiple messages concurrently, but only a single receiver consumes any one message.

Publish-Subscribe Channel

Problem:

The service provider broadcasts an event once, to all interested consumers.

Solution:

Send the event on a *publish-subscribe* channel, which delivers a copy of a particular event to each receiver. See *Figure 8*.

Interactions:

A *publish-subscribe channel* that is developed based on Observer pattern [9], and describes a single input channel that splits into multiple output channels – one for each subscriber. After publishing an event into the *publish-subscribe channel*, the same message is delivered to each of the output channels. Each output channel is configured on one-to-one topology to allow only one consumer to consume a message. The event is considered consumed only when all of the consumers have been notified.

A *publish-subscribe channel* can be a useful for systems management, error debugging and different level of testing.

Datatype Channel

Problem:

The receiver must know what type of messages it is receiving, or it won't know how to process them. For example, a sender might send different objects such as purchase orders, price quotes, and queries, but a receiver will probably take different steps to process each of these, so it has to know which is which.

Solution:

Use a separate *datatype channel* for each data type, so that all data on a particular channel is of the same type. See *Figure 9*.

Interactions:

In any messaging system there are several separate *datatype channels* for each type of data. All of the messages on a given channel will contain the same type of data. Based on data type, the service provider sends the data to the channel and the consumer receives data from the appropriate *datatype channel*.

Figure 7. Point-to-point Channel

Figure 8. Publish-Subscribe Channel

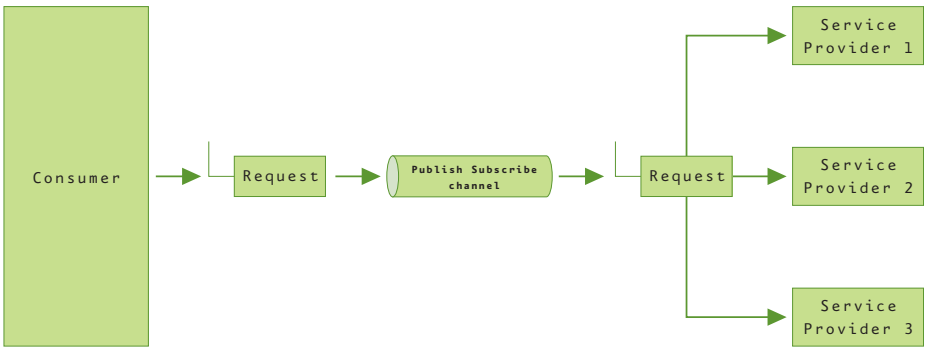


Figure 9. Datatype Channel

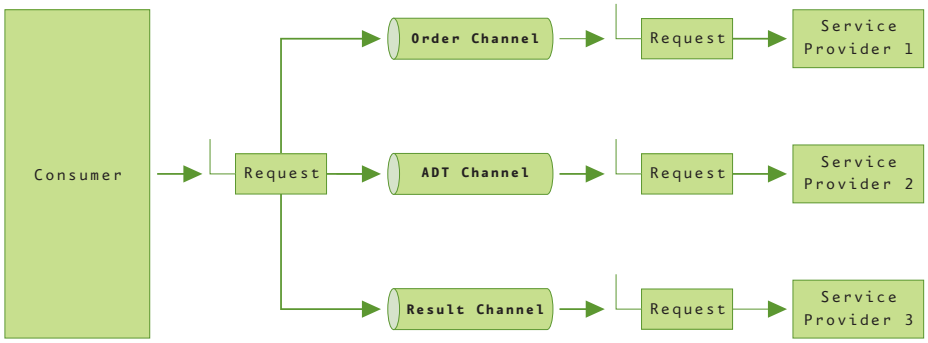
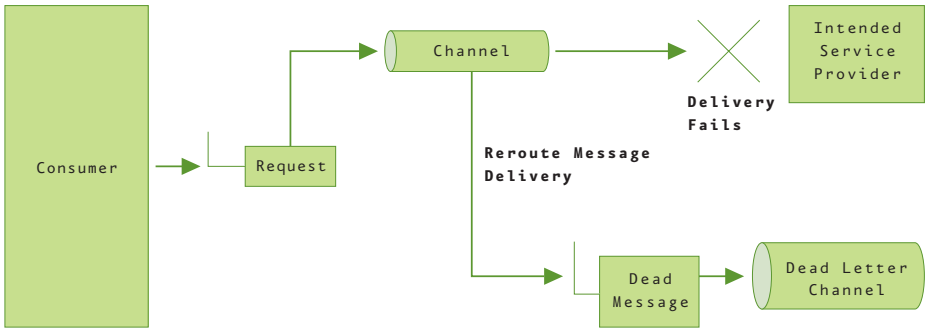


Figure 10. Dead Letter Channel



Dead Letter Channel

Problem:

There are a number of reasons for message delivery to fail. Issues might be message channel configuration problem, a problem with consumers, or message expiration.

Solution:

When there is any delivery issue with the message, it can be moved to a different messaging channel called a *dead letter channel*. See *Figure 10*.

Interactions:

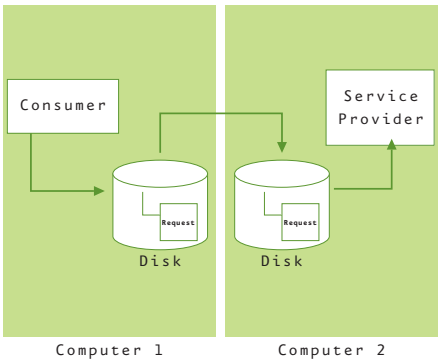
A *dead letter channel* is a separate channel dedicated for bad messages or invalid messages. From this channel messages can be rerouted to the mainstream channel or even in separate channel for special processing of the message.

Guaranteed Delivery

Problem:

One of the main advantages of asynchronous messaging over RPC is that the participants don't need to be online at the same time. While the network is unavailable, the messaging system has to use a store and forward mechanism to ensure message durability. By default, the messages are stored in memory until they can be successfully forwarded to the next contract point. This mechanism works well when the messaging system is running reliably, but if the messaging system crashes, all of the stored messages are lost. As a preventative measure, applications use persistent media like files and databases to ensure recovery from system crashes.

Figure 11. Guaranteed Delivery



Solution:

Use a *guaranteed delivery* mechanism to make messages persistent. See *Figure 11*.

Interactions:

With *guaranteed delivery*, the messaging system uses a built-in data store (local storage disk space in a participant computer) to persist messages in each participant computer on which the messaging system is installed. The message is safely stored until it is successfully delivered. In this way, it ensures guaranteed delivery.

This guaranteed delivery mechanism increases system reliability, but at the expense of performance as it involves considerable numbers of I/O and consumes a large amount of disk space. Therefore if performance or debugging/ testing is the priority try to avoid using *guaranteed delivery*.

Message Bus

Problem:

An enterprise consists of various

independent applications communicating with each other in a unified manner. We need an integration/service architecture that enables those applications to coordinate in a loosely coupled fashioned.

Solution:
Structure the connecting middleware between these applications as a *message bus* that enables them to work together using messaging as shown in *Figure 12*.

Interactions:
A *message bus* is a combination of a common data model, a common command set, and a messaging infrastructure to allow different heterogeneous systems to communicate through a shared set of interfaces.

A *message bus* can be considered as a universal connector between the various enterprise systems, and as a universal interface for client applications that wish to communicate with each other.

A *message bus* requires that all of the applications should use the same canonical data model. Applications adding messages to the bus may need to depend on message routers to route the messages to the appropriate final destinations.

Message Routing Patterns
Almost all messaging system uses built in router as well as customized routing. Message Routers are very important building blocks for any good integration architecture. As opposed to the different message routing design patterns, this pattern describes a hub-and-spoke architectural style with few specially embedded routing logic.

Figure 12. Message Bus

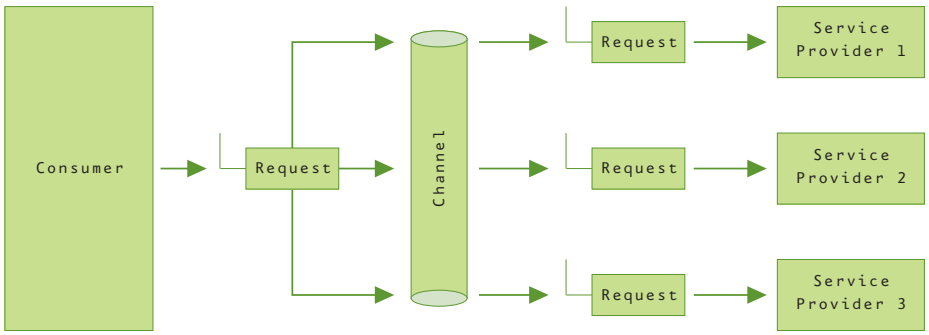
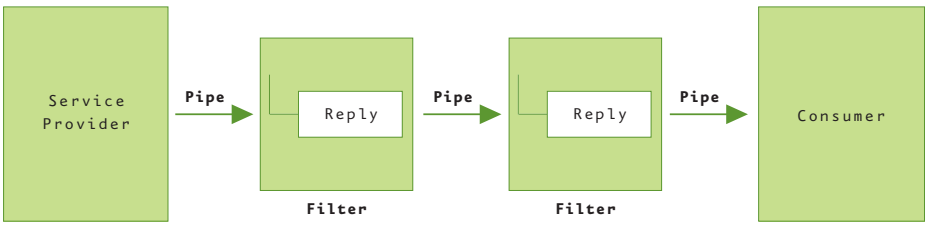


Figure 13. Pipes and Filter



In search of the right router
An important decision for an architect is to choose the appropriate routing mechanism. Patterns that will help you make the right decision are:
– Pipes and Filter
– Content-Based Router
– Content Aggregator

Pipes and Filter
Problem:
How can you divide a larger processing task into a sequence of smaller, independent processing steps?

Solution:
Use the *pipes and filters* pattern to divide a larger processing task into a sequence of smaller, independent processing steps (filters) that are connected by channels (pipes).
See *Figure 13*.

Interactions:
Each filter exposes a very simple interface: it receives messages on the inbound pipe, processes the message, performs business transformations, and publishes the results to the outbound pipe. The pipe connects one filter to the next, sending output messages from one filter to the next. It's very similar to execution of a method call through passing parameters and getting a return value. It follows 'chain of responsibility' [11] pattern. Because all components use the same external interface they can be composed into different solutions by connecting the components to different pipes. The connection between filter and pipe is sometimes called port. In the basic form, each filter component has one input port and one output port.

The *pipes and filters* pattern uses abstract pipes to decouple components from each other. The pipe allows one component to send a message into the pipe so that it can be consumed later by another process that is unknown to the component. One of the potential downsides of pipes and filters architecture is the larger number of required channels that consume memory and CPU cycles. Also, publishing a message to a channel involves a certain amount of overhead because the data has to be translated from the application-internal format into the messaging infrastructure's own format.

Using *pipes and filters* also improves module-wise unit testing ability. It can help to prepare a testing framework. It is more efficient to test and debug each core function in isolation because we can tailor the test mechanism to the specific function.

Content-Based Router

Problem:

The routing can be based on a number of criteria such as existence of fields, specific field values, and so on.

Solution:

Use a content-based router to route each message to correct consumer based on message content. See *Figure 14*.

Interactions:

The *content-based router* examines the message content and routes the message onto a different channel based on message data content. When implementing a *content-based router*, special caution should be taken to make easily maintainable routing logic. In more sophisticated integration scenarios, the content-based router can be implemented as a configurable rules engine that computes the destination channel based on a set of configurable rules.

Content Aggregator

Problem:

The messaging system exchanges messages between a variety of sources. The messages have similar content but different formats, which can complicate the processing of combined messages. It would be better processing decision if we assigned different components with different responsibilities [11]. For example, if we want to select all of the transactions of a particular customer from different business zones for a particular quarter. This method is called event linking and sequencing.

Solution:

Use a stateful *content aggregator*, to collect and store individual messages and combine those related messages to publish a single aggregated message. See *Figure 15*.

Interactions:

A *content aggregator* is a special filter that receives a stream of messages and correlates related messages. When a complete set of messages has been received, the aggregator collects information from each correlated message and publishes a single, aggregated message to the output channel for further processing. Therefore, aggregator has to be stateful, because it needs to save the message state with processing state until the complete formation of the aggregation.

Figure 14. Content Based Router

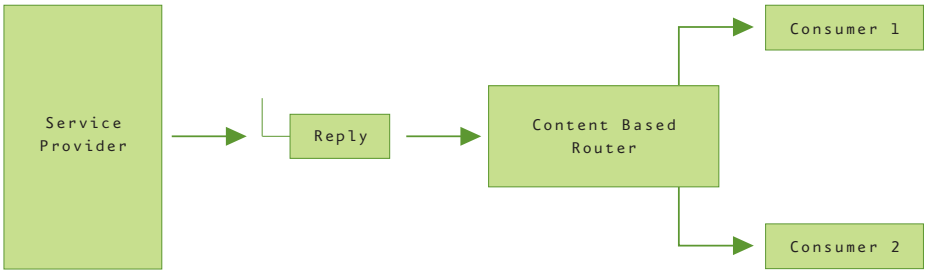
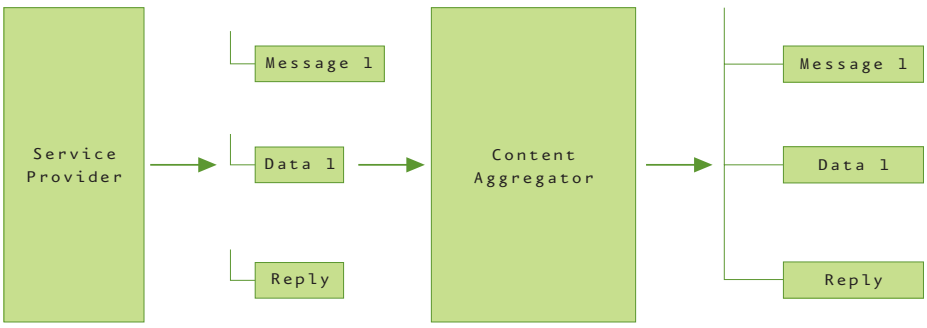


Figure 15. Content Aggregator



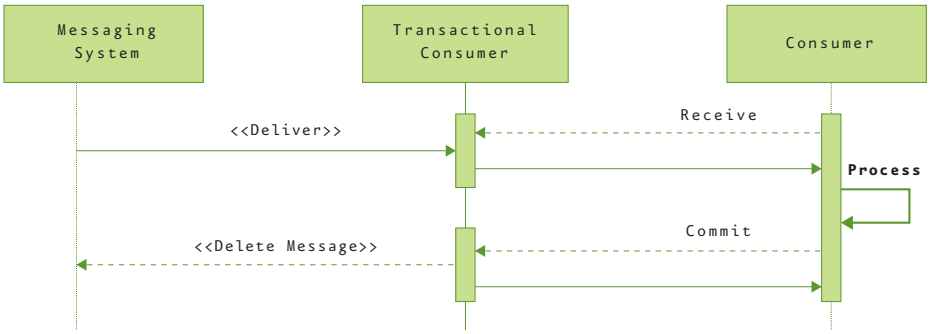
When designing an aggregator, we need to specify the following items:
– *Correlation Id* – An identifier that indicates messages internal relationship
– *End Condition* – The condition that determines when to stop processing
– *Aggregation Algorithm* – The algorithm used to combine the received messages into a single output message

Every time the *content aggregator* receives a new message, it checks whether the message is a part of already existing aggregate or a new aggregate. After adding the message, the *content aggregator* evaluates the process end condition for the aggregate. If the condition evaluates to true, a new aggregated message is formed from the aggregate and published to the output channel. If the process end condition evaluates to false, no message is published and the *content aggregator* continues processing.

Service Consumer Patterns
There are several possible types of Service Consumer. In this pattern catalogue we will present a set of consumer patterns.

Transactional Client
Problem:
Transactions are important part of any messaging system and are sufficient for any participant to send or receive a single message. However, a few specific scenarios might need a broader transactional approach, which in turn may need special transactional coordination. These cases include (but are not limited to):
– *Send-Receive Message Pairs* – Receive one message and send another.

Figure 16. Transactional Client Sequence Diagram



– *Batch Message* – Send or receive a group of related messages in a batch mode.
– *Message / Database Coordination* – Send or receive a message combined with database update. For example, when an application receives and processes a message for ordering a product, the application will also need to update the product inventory database.

Scenarios like these require a different specification of transactional boundaries with much more complexities involving more than just a single message and may involve other transactional stores besides the messaging system.

How can you solve this kind of transactional problems?

Solution:
Use a *transactional client* – make the client's session with the messaging system transactional and ensure that the client can specify complex transaction boundaries. See *Figure 16*.

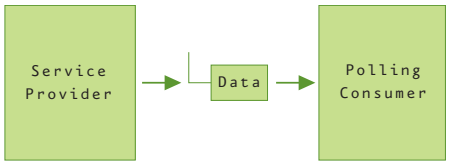
Interactions:
Both participants can be transactional. From a sender's point of view, the message isn't considered added to the channel until the sender commits the transaction. On the other hand message isn't removed from the channel until the receiver commits the transaction.

With a transactional receiver, messages can be received without actual removal of the message from the channel. The advantage of this approach is that if the application crashed at this point, the message would still be on the queue after message recovery has been performed; the message would not be lost. After the message processing is finished, and on successful transaction commit, the message is removed from the channel.

Polling Consumer
Problem:
In any messaging system, the consumer needs an indication that application is ready so that it can consume the message.

The best approach for the consumer is to repeatedly check the channel for message availability. If any message is available, the consumer consumes it. This checking is a continuous process known as polling.

Figure 17. Polling Consumer



Solution:
The application should use a *polling consumer*, one that explicitly makes a call when it wants to receive a message. See *Figure 17*.

Interactions:
A *polling consumer* is a message receiver. A *polling consumer* restricts the number of concurrent messages to be consumed by limiting the number of polling threads. In this way, it prevents the application from being blocked by having to process too many requests, and keeps any extra messages queued up until the receiver can process them.

Event-Driven Consumer
Problem:
The problem with polling consumers is that it's a continuous process involves dedicated threads and consumes process time while polling for messages.

Solution:
Instead of making the consumer poll for the message, a better idea might be to use event driven message notifications to indicate message availability. See *Figure 18* and *Figure 19*.

Figure 18. Event-Driven Consumer

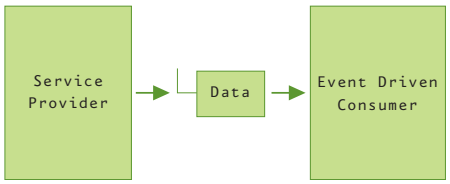
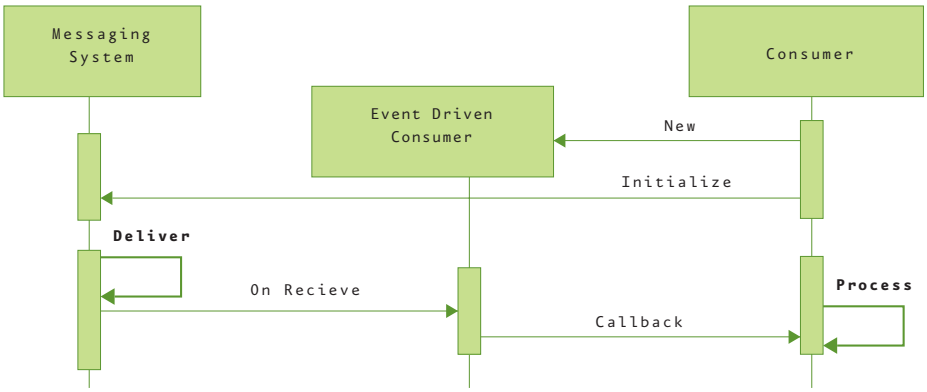


Figure 19. Event-Driven Consumer Sequence Diagram



The application should use an event-driven consumer. Event-driven consumers automatically consume messages as they become available.

Interactions:
An event-driven consumer is invoked by the messaging system at the time of message arrival on the consumer's channel. The consumer uses application-specific callback mechanism to pass the message to the application.

Durable Subscriber
Problem:
In some cases you might require guaranteed message delivery where a message consumer is not connected to publish-subscribe channel or has crashed before receiving a message. In this case the messaging system needs to ensure guaranteed message delivery when the consumer reconnects to the system.

Solution:
Use a durable subscriber. See *Figure 20 and 21*.

Interactions:
A durable subscription saves messages for an off-line subscriber and ensures message delivery when the subscriber reconnects. Thus it prevents published messages from getting lost and ensures guaranteed delivery. A durable subscription has no effect on the normal behavior of the online/active subscription mechanism.

Figure 20. Durable Subscriber

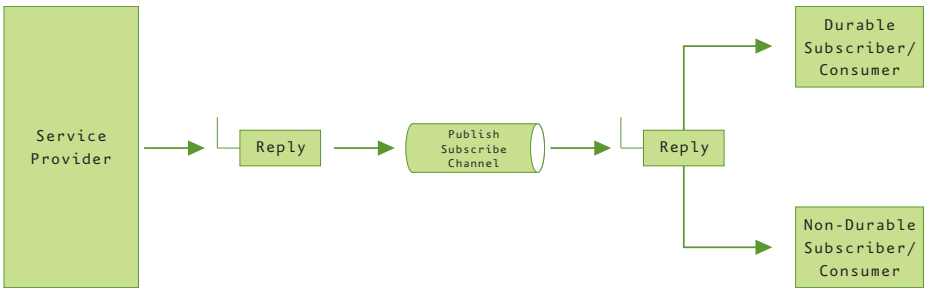
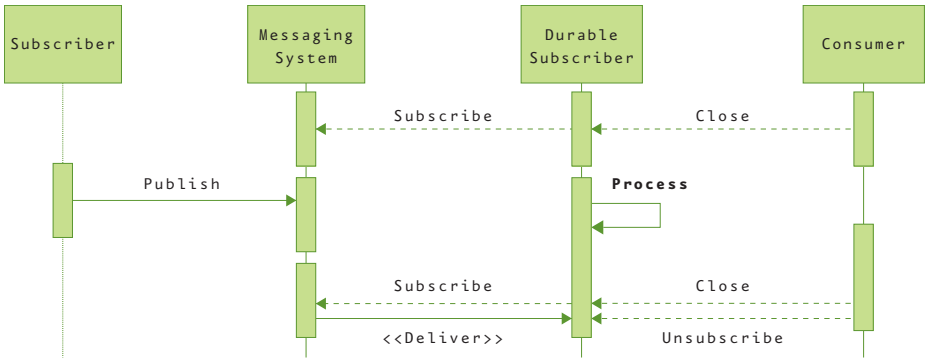


Figure 21. Durable Subscriber sequence diagram



Idempotent Receiver

Problem:

For certain scenarios, instead of using Durable Subscription mechanism, some reliable messaging implementations can produce duplicate messages to ensure guaranteed, at-least once Delivery. In these cases, message delivery can generally only be guaranteed by resending the message until an acknowledgment is returned from the recipient. However, if the acknowledgment is lost due to an unreliable connection, the sender may resend a message that the receiver has already received.

We need to ensure that the messaging system is able to safely handle any messages that are received multiple times.

Solution:

Design a receiver to be an idempotent receiver. See *Figure 22*.

Interactions:

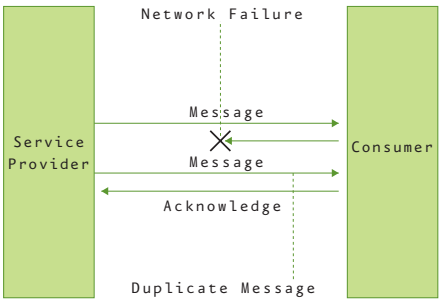
The term idempotent is originated from mathematics to describe the ability of a function that produces the same result if it is applied to itself, i.e. f(x) = f(f(x)).

In messaging Environment this concept ensures safely resent of same message irrespective of receipt of same message multiple times.

In order to detect and eliminate duplicate messages based on the message identifier, the message consumer has to maintain a buffer of already received message identifiers. One of the key design issues is to decide message persisting timeout. In the simplest case, the service provider sends one message at a time, awaiting the receiver's acknowledgment after every message. In this scenario, the consumer efficiently uses the message identifier to check that the identifiers are identical. In practice, this style of communication is very inefficient, especially when significant throughput is required. In these situations, the sender might want to send a whole set of messages in a batch mode without awaiting acknowledgment for individual one. This will necessitate keeping a longer history of identifiers for already received messages, and the size of the message subscriber's buffer will grow significantly depending on the number of message the sender can send without an acknowledgment.

An alternative approach to achieving idempotency is to define the semantics of a message such that resending the message does not impact the system. For example, rather than defining a message as variable equation like 'Add 0.3% commission to the Employee code A1000 having a base salary of $10 000', we could change the message to 'Set the commission amount $300.00 to the Employee code A1000 having a base salary of $10 000'. Both messages achieve the same result – even if the current commission is $300. The second message is idempotent because receiving it twice will not have any effect. So whenever possible, try to sendconstants as message and avoid variables in messages. In this way we can efficiently achieve idempotency.

Figure 22. Duplicate Message Problem

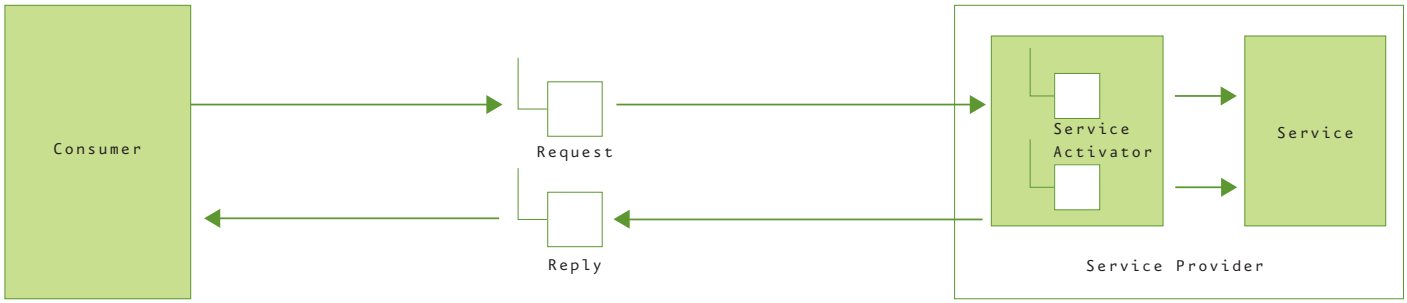


Service Factory

Problem:

When designing service consumer for multiple styles of communication, it might seem necessary to define the service for each style, and this concept can be linked to the Factory Design Pattern [9]. In SOA it's a challenge to invoke the right services based on the style of communication.

Figure 23. Service Factory



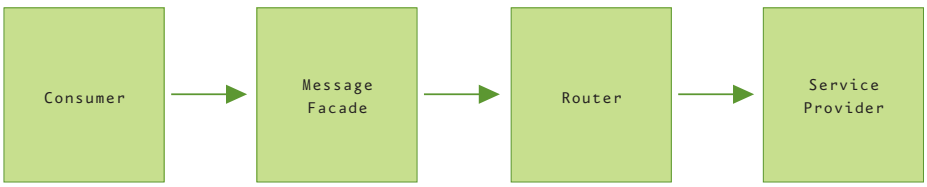
Solution:
Design a *service factory* that connects the messages on the channel to the service being accessed. See *Figure 23*.

Interactions:
A *service factory* may return a simple method call or even a complex remote process invocation. The *service factory* invokes the service just like any other method invocation and optionally can create a customized reply message.

Message Façade Pattern
Problem:
Depending on business requirements, you might need to encapsulate business logic flow and complexity behind a standard façade.

Solution:
A message façade can be used asynchronously and maintained independently. It acts as an interceptor between service consumer and service provider. See *Figure 24*.

Figure 24. Message Façade



Interactions:
The client creates a command message and sends it to the message façade through messaging channel. The façade receives the message (using a polling consumer or an event-driven consumer) and uses the information it contains to access business tier code to fulfill a use case. Optionally, a return message is sent to the client confirming successful completion of the use case and returning data.

Conclusion
So far we have understood how messaging patterns exist at different levels of abstraction in SOA. In this paper, which is the first of a two-part series on messaging patterns in service oriented architecture, Message Type Patterns were used to describe different varieties of messages in SOA, Message Channel Patterns explained messaging transport systems, Routing Patterns explained mechanisms to route messages between the Service Provider and Service Consumer, and finally Service Consumer Patterns illustrated the behavior of messaging system clients. In the next issue of JOURNAL, the final part of this paper will cover Contract Patterns that illustrate the behavioral specifications required to maintain smooth communications between Service Provider and Service Consumer and Message Construction Patterns that describe creation of message content that travels across the messaging system.

Copyright Declaration
G Hohpe & B Woolf, *Enterprise Integration Patterns*, (adapted material from pages 59-83), (c) 2004 Pearson Education, Inc. Reproduced by permission of Pearson Education, Inc. Publishing as Pearson Addison Wesley. All rights reserved.

References

1. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Gregor Hohpe and Bobby Woolf, Addison-Wesley, 2004
2. *Service Oriented architecture: A Primer*, Michael S Pallos, EAI Journal, December 2001
3. *Solving Information Integration Challenges in a Service-Oriented Enterprise*, ZapThink Whitepaper, http://www.zapthink.com
4. *SOA and EAI*, De Gamma Website, http://www.2gamma.com/en/produit/soa/eai.asp
5. *Introduction to Service-Oriented Programming*, Guy Bieber and Jeff Carpenter, Project Openwings, Motorola ISD, 2002
6. *Java Web Services Architecture*, James McGovern, Sameer Tyagi, Michael Stevens, and Sunil Mathew, Morgan Kaufman Press, 2003
7. *Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications*, Alan Brown, Simon Johnston, and Kevin Kelly, IBM, June 2003
8. *The Modular Structure of Complex Systems*, Parnas D and Clements P, IEEE Journal, 1984
9. *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma E, Helm R, Johnson R, and Vlissides J, Addison-Wesley, 1994
10. *Computerworld Year-End Special: 2004 Unplugged*, Vol. 10, Issue No. 10, 15 December 2003–6 January 2004, http://www.computerworld.com.sg/pcwsg.nsf/currentfp/fp
11. *Applying UML and Patterns – An introduction to OOA/D and the Unified Process*, Craig Larman, 2001

Soumen Chatterjee,
Senior Consultant, CGE&Y (India)
soumen.chatterjee@cgey.com

Soumen is a Microsoft Certified Professional and Sun Certified Enterprise Architect. He's significantly involved in enterprise application integration and distributed object oriented system development using Java/J2EE technology to serve global giants in the finance and health care industries. With expertise in EAI design patterns, messaging patterns and testing strategies he designs and develops scalable, reusable, maintainable and performance tuned EAI architectures. Soumen is a Senior Consultant with Cap Gemini Ernst & Young. He's an admirer of extreme programming methodology and has primary interests in AOP and EAI. Besides software Soumen likes movies, music and follows mind power technologies.

JOURNAL2

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

Managing Editor
Graeme Malcolm
Principal Technologist, Content Master Ltd

Editorial Board
Christopher Baldwin
Principal Consultant, Developer and Platform Evangelism Group, Microsoft EMEA
Gianpaolo Carraro
Architect, Developer and Platform Evangelism Group, Microsoft EMEA
Simon Guest
Program Manager, PSPG Architecture Strategy, Microsoft Corporation
http://www.simonguest.com
Wilfried Grommen
General Manager, Business Strategy, Microsoft EMEA
Richard Hughes
Program Manager, PSPG Architecture Strategy Microsoft Corporation
Neil Hutson
Director of Windows Evangelism, Platform Strategy and Partner Group, Microsoft Corporation
Eugenio Pace
Principal Consultant, Microsoft Consulting Services, Microsoft Argentina
Harry Pierson
Architect, PSPG Architecture Strategy, Microsoft Corporation
http://devhawk.com
Michael Platt
Architect, Developer and Platform Evangelism Group, Microsoft Ltd
http://blogs.msdn.com/michael_platt
Philip Teale
Partner Strategy Manager, Enterprise Partner Group, Microsoft Ltd

Project Management
Content Master Ltd
www.contentmaster.com

Design Direction
venturethree, London
www.venturethree.com

Orchestration
Katharine Pike
WW Architect Programs Manager, PSPG Architecture Strategy, Microsoft Corporation

Foreword Contributor
Michael Platt
Architect, Developer and Platform Evangelism Group, Microsoft Ltd
http://blogs.msdn.com/michael_platt

Microsoft®

Microsoft is a registered trademark of Microsoft Corporation

The information contained in this Microsoft® 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.

All copyright, trade marks and other intellectual property rights in the material contained in the Journal belong, or are licenced to, Microsoft Corporation. Copyright © 2003 All rights reserved. You may not copy, reproduce, transmit, store, adapt or modify the layout or content of this Journal without the prior written consent of Microsoft Corporation and the individual authors. Unless otherwise specified, the authors of the literary and artistic works in this Journal have asserted their moral right pursuant to Section 77 of the Copyright Designs and Patents Act 1988 to be identified as the author of those works.