CHAPTER

2

Service Oriented Architecture and Software Components

with Derek Andrews

It is to be all made of sighs and tears; –
It is to be all made of faith and service; –
William Shakespeare (*As You Like It*)

In this chapter we digress slightly, in order to locate the business rules approach, within the context of service oriented architecture (SOA), the modern, more flexible approach to structuring decentralized IT systems. We will see that many of the ideas and techniques of SOA carry over directly to the business rules approach, and the adoption of SOA and BRMS together will let businesses deploy or integrate new applications far more easily. One of the key points of this chapter is that SOA, like business rules, is about raising the abstraction levels of interfaces: interfaces that must support the business, not the system.

The examination of any engineering discipline must start by looking at its history, in the hope of determining trends and learning from the mistakes of the past. Programming computers used to be easy, although producing high quality software was never so and remains a key challenge. Over five decades, programming has evolved into a much more complex activity, and the goal of producing good software is as elusive as ever.

In the 1950s, it was a simple matter to persuade an electronic brain (as they were then dubbed) to add up some numbers and print them out or, provided that you knew the mathematics, compute the cosine of an angle. The invention of high level languages such as FORTRAN, ALGOL and COBOL made such tasks even simpler. However, there was a double price to pay. Firstly, you had to choose the right language for the job; COBOL doesn't

know much about trigonometry. Secondly, the poor programmer has to know that the COS function is in the language's library. Then came databases, removing the need to worry about much of the complexity of data storage and management – provided that you knew the language and, indeed, the theoretical basis of the database. At the same time we saw the advent of fourth generation languages (4GLs) that made programming easier still. But, here too, there was a price; such languages were predicated on relatively fixed data structures. Just as we wouldn't choose COBOL to calculate sines, we would not choose a spreadsheet to write a computer game.

By the mid 1990s, it was clear that the future of computing was synonymous with the idea of distributed computing. Life suddenly got much harder. Now the programmer not only programs, but does so within an environment that requires the use of a bewildering array of pre-written components supplied within an architectural framework such as .NET or J2EE. During this evolution, the dream of the 4GL largely evaporated, outside of the world of database management anyway. 4GL programming was largely superseded by programming in object-oriented 3GLs such as C++, C# or Java.

The move to component architectures raised the stakes in terms of software design too. The complexity itself means that good modelling of the logical structure of systems becomes critical. The banner of component-based design (CBD) is raised in the battlefield of computing. Szyperski's (1998) groundbreaking book told us how to program with components. Sims (1994), D'Souza and Wills (1999), and later Cheesman and Daniels (2000), began to show us how to design for component frameworks. Much of this work seemed to say that CBD was a major advance over object-oriented methods and superseded the latter, but there were dissenting voices (including my own) that took the view that earlier work had merely deviated from pure object-oriented principles and that CBD was merely OO 'done right' and, in fact, the current commercial frameworks were encouraging a very non-OO approach to design by separating coherent business objects into process layers, data layers, etc. Typical of such dissenters were Pawson and Matthews (2002).

Add to all this the emergence of new technologies such as grid computing, peer-to-peer computing, web services and agent-based systems, and we have a recipe for a goulash of staggering complexity.

The only way to integrate these views and maintain good design disciplines seems to be to regard components and, indeed, all objects as suppliers or consumers of services. Indeed, systems based on a service oriented architecture (SOA) could be said to be the natural next step in software development. If we look back again at the history of software development, we see at the start that programs were written to solve a particular scientific problem; next programs were written to help with business: programs were written to read, process and put data out in a batch environment. Online systems followed; these were for clerks to work with, usually driving a legacy system adapted from batch to handle work in real time. The introduction of the world wide web led to online

systems for customers who could then use a system directly, rather than using it indirectly through a clerk. The natural consequence is the online system for anybody (customers, people, partners, other businesses), not necessarily with any human computer interface of the old type.

Business rules management technology dovetails neatly – and essentially – into the SOA approach. The change in philosophy needed to produce systems that support the business is leading to new business opportunities, a different way of developing systems and doing business. However, if our main goal is to align IT better with business, then SOA alone is not enough. We need to separate the business rules from the code to achieve this. In other words, SOA without BRMS is like a runaway train with no wheels; it will soon grind to an ignominious halt.

We now look at the philosophy of SOA and outline the business drivers for and the benefits and pitfalls of adopting it.

2.1 Service Oriented Architecture and Business Rules

Service oriented architecture is an architectural concept in software design that emphasizes the use of combined services to support business requirements. In SOA, resources are made available to service consumers in the network as independent artifacts that are accessed in a standardized way. Many definitions of SOA identify it with the use of web services using standards such as SOAP (originally Simple Object Access Protocol) and WSDL (Web Services Description Language – pronounced *wuhzdle*). However, it is possible to implement SOA using any service-based technology. Though built on similar principles, SOA is **not** coextensive with web services, the latter being a collection of technologies and standards, such as SOAP and XML. The notion of SOA is quite independent of any specific technology.

Critical to this notion of services is their loosely coupled character; service interfaces are independent of their implementations. Application developers or system integrators can build applications by composing one or more services without having to know the services' underlying implementations. For example, a service can be implemented either in a .NET or J2EE environment, and the application consuming the service can even run on a different platform or be written in a different language.

Consider, for example, someone enquiring about the parts needed to construct a floggle and their costs. The service might respond with something like 'You need six widgets @ 6p and one 6 mm toggle @ £1.20. The total cost is £1.56. All items are currently in stock.'

The service meets the need of this type of customer well, but it should be clear that there are at least three underlying services, concerned with bills of

materials, pricing and stock. It is almost certainly more flexible to implement these services as separate components and aggregate them into higher level services like the one described.

We can note three further features of SOA:

- SOA services have self-describing interfaces in platform-independent documents. In the case of web services, these documents are presented in XML, and WSDL is the standard used to describe the services.
- SOA services communicate with messages using a formally defined language. Consumers and providers of services typically exist in heterogeneous environments, and consumers communicate with the least possible knowledge about their provider. Messages between services can be viewed as if they were business documents. In the case of web services, communication is via XML schemata (also called XSD).
- Ideally, SOA services are maintained in a registry that acts as a directory listing. Applications can then look up the services needed in the registry and invoke them as required. In the case of web services, Universal Description, Definition, and Integration (UDDI) is the standard used for service registry definition.

Each SOA service may have a quality of service (QoS) associated with it. Typical QoS elements include security requirements, such as authentication and authorization, reliable messaging, and policies regarding who can invoke services. However, more business-oriented service level agreements can also be important. Consider a financial pricing service that gives current stock prices based on current trading in the market. There are two components that implement the same service interface. One is from Reuters – a well-established and reputable vendor – and the other is supplied by Honest John's Prices Inc. Do you care which implementation you take? Of course you do, if reliability is an issue. But SOA is implementation independent, so you shouldn't have to care. The solution is to include a QoS factor in the service interface that measures the 'reputation' of the supplier[1]. Of course, when the services are provided in house the quality may be inferred, and these considerations do not apply.

Service oriented architecture structures software systems in the following style:

- Distributed enterprise application servers provide a collection of services (or transactions).
- By various incentives (which may include quarterly bonuses), developers are encouraged to build systems using this collection of services to supply most of the functions of new applications; roughly speaking we assemble new applications by plugging together existing services.

[1]We are indebted to John Daniels for this example.

- These units of transaction can be relocated, load-balanced, replaced, security-applied, etc.

One of the ideas behind component based development is to scale up the object oriented philosophy of encapsulation, interfaces and polymorphism to the component level – a component is just a big object, designed and developed with the same care and attention given to identifying classes, their responsibilities and their collaborations. This approach can be further extended to a service oriented system by dividing the system into a set of components, each of which supplies a set of business services. Done intelligently, this leads to the system being built from a set of loosely connected components, many of which are ripe for reuse, or even better – sharing.

Because of the nature of components, we can try and factor other decisions into their design as well as just responsibilities and collaborations. If we try for a layered architecture (always a good idea) at the lowest level we can identify components that supply business utility services. These components supply utility services that are useful across a family of businesses. For example an address book, a catalogue, or a component that deals with interest rates. This type of component encapsulates few or no business rules, and can be categorized as being 'function-like'. By function-like we mean that they behave as a look-up table might, indexed by a key, or as a mathematical function – a particular input will (nearly) always supply the same answer. A key such as zip code and building number should yield a unique address. A book catalogue should, for a given ISBN, produce the details of the relevant book, and such details do (or should) not change. For interest rate calculations, given a period, interest rate and principle amount; the result should not change (a real interest rate calculator component would not be that simple, but the principle remains the same). These components are, by their nature, very stable and should be reusable within a particular business area. This type of component extends the old idea of a FORTRAN code library, but brings it up-to-date.

At the next level up are components that encapsulate business objects: parties, places or things. These components would, for example, manage customers (or even better people or companies, one of whose rôles is customer), locations for a parcel delivery company, or book copies. These components are still relatively stable in the sense that once written, they tend to have only minor modifications made to them later: usually the need to add additional information or roles. They are re-usable, or shareable, within a particular company. The interface to them is basically Create-Read-Update-Delete, suitably renamed – for a library member the services would be join and resign, lots of query operations to do with being a member of a library and operations to change details, such as a library member's address or their name if this is requested when they get married. It may look as if such components might only encapsulate a single type; but even in a simple example, the component managing members in a library system may

have as many as 10 different types in it, and a component containing customer details would be very much larger. Business rules are usually found in these components – hence reusability is likely to be restricted to a particular business area.

At the top level we can define components that manage business processes; these contain objects that record events such as a book loan or a reservation for a title. Business objects will play particular rôles in a business process event, and thus appear in the record of that event. For example, in a library a person plays the rôle of a borrower in a book loan and a reserver in a reservation and as a library member in the library. The rôles link to, and represent, business objects. Business rules about the processes can be held in the corresponding process component. The component managing loans would know about the length of a loan, and the maximum number of books that can be borrowed at one time. A process component is less reusable than the other two types as it contains business rules that govern the process. These components are less stable; they tend to change quite frequently as the organization thinks of better and different ways of conducting its business. Even these can be made easier to write and maintain if they can be split into two parts, a generic description of a particular business process together with a part that tailors the description to a particular business process by providing the business rules the restrict the general approach. This latter part should be written and developed as a plug-in. Returning to the library example, the loan component is about lending books to library members, but with care can be refactored to be used in any organization that does loans (books become lendables and library members borrowers). The business rules about the business process can be turned into plug-ins and can be changed as needed.

Looking at any business process, one part is about the order in which we do things, and the other is about under which conditions we do things: you can only reserve a horror movie if you are over 18, you cannot buy life-insurance if you are over 100, you cannot have a loan if you are an un-discharged bankrupt. These later business rules can be encapsulated in a separate part of the component. To introduce generality into the first part we can allow more general order in which we do things and impose business rule to restrict this. Consider the business process of making a sale; our business rules may demand a prescribed order for the activities that make up a sale as shown in Figure 2.1 (a). This ordering could be weakened to that illustrated in Figure 2.1 (b).

The latter allows the activities that make up the business process to be done in any order, or even in parallel. Consider a Christmas club, where we pay for goods before we finally obtain them, and a ‘try-before-you-buy’, where they are delivered before being ordered. Business rules are used to enforce the previous order for new or unreliable customers; reliable and well-known customers could benefit from more liberal régimes.

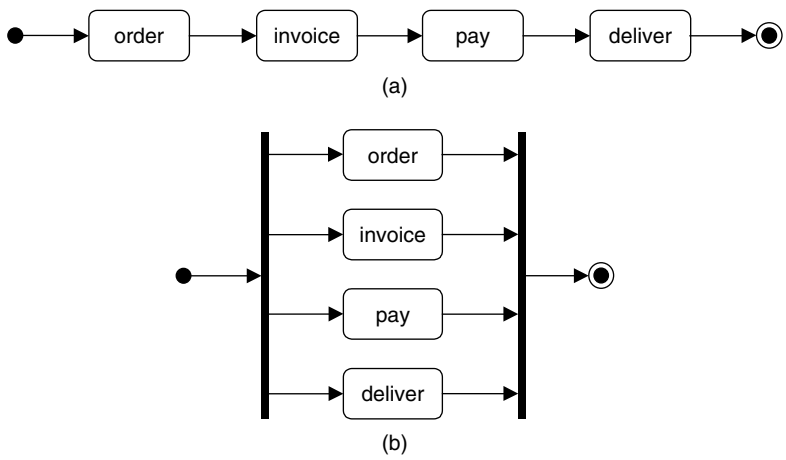

Figure 2-1 Sales processes.

With the breakdown into three types of component[2], it is now possible to examine the build, buy or wrap decision. COTS (Commercial Off-the-Shelf) components are more likely to be utility or business object components, since these components include few or no business rules, and are thus likely to be more general. New application areas managing business processes in an organization are more likely to be built in house rather than bought off the shelf, as these are likely to include rules which are peculiar to the organization. Legacy systems tend not to have layers corresponding to the three component types, but have process, objects, rules and utility services mixed together. Wrapping legacy systems in a component is wrapping legacy business processes and rules – a sure way of perpetuating old-fashioned ways of doing business. Wrapping the legacy at the utilities or business object level, if these types of services can be teased out, is a better idea. Wrapping at the business object component level is necessary; wrapping legacy at the business process layer is difficult and is also wrong; it is most likely that this is the part that will change the most. Old stovepipe systems encapsulate their own business rules, and these rules may conflict with other stovepipes, so trying to combine them is likely to lead to problems – an insurance company may wish to offer a package of mortgage, term insurance convertible to life insurance and medical insurance, and an endowment to pay off the mortgage, and might find that the legacy rules are such that only one person in a million is eligible; this package may need some weakening of the individual rules for each financial package.

[2]This discussion refers implicitly to Peter Coad's (1999) 'colour' patterns, where he distinguishes the recurring object types: PartyPlaceThing (green), Rôle (yellow) and MomentInterval (pink). We prefer to call the latter type Episode. Coad's insights are core to one of the patterns in Chapter 7: BUILD A TYPE MODEL (4).

In the long run, if the company is made up from a lot of different companies that have merged, then there may be many versions of the same core component (e.g. customers of the different companies). A strategy needs to be developed to merge all of these into a group of components, each covering a core type. The first thing to do is to get the business rules out of the legacy systems so it is only data that are being merged – not data plus business rules.

Having said all this, it must be admitted that it is just as easy to construct a weak or dysfunctional SOA as it is to mess up any other kind of computer system. Also, quality is much more of a problem; any errors will be visible globally. If your services are no good, your reputation will suffer and your services will lie unused. Worse still, badly dysfunctional services may attract litigation. Another danger is basing a SOA on proprietary product line architecture and thus preventing unforeseen variation.

The route to SOA involves supporting business goals by supplying services to the users of the system so that business can be conducted more easily; it involves the infrastructure to support those services, both from the application and its platform. To do this, it is necessary to understand the business. This cannot be done by just looking at the system users and their use cases; the net must be cast wider. To supply a set of services it is necessary to understand the business and find the real users of the system, and these are not usually the people sitting in front of a keyboard and screen. The real users of the system want to achieve a goal, and a computer system is just a tool to help to achieve that goal. SOA is about designing and developing systems that supply services fit for the purpose of helping users attain their goals.

Decentralized computing provides greatly increased flexibility for both business and IT, but it also creates a danger: business decisions may be inconsistent across applications. BRMS provide a way of managing the decision logic centrally and independently, even where the rules apply locally to different 'zones' within an organization. The key is the use of a repository to store the rules and a rule engine to apply them correctly in the context of a particular application or service. Thus a BRMS can act as a mediator between service oriented applications and the legacy; it smooths the interactions between applications and acts as the decision management component for applications that are implemented as a set of services. These services interact with a decision management service which incorporates a rule engine; this, in turn, has access to the repository. As the intent of each service is distinct, so too is the service's use of rules. Two services might use a common rule, but both may have unique rules or processes related to that rule. Using a BRMS allows reuse of rules across services. This, in turn, can speed development and ease maintenance even more than the adoption of SOA on its own, precisely because the services are easier to configure; when the business changes the rules can be changed without need to recast the services or architecture at the code level.

Service oriented architectures and business rules management systems are an essential component of modern agile businesses. They vastly reduce

the problems associated with the evolution of complex and volatile business strategies and policies. SOA and BRMS are parallel and complimentary technologies.

A good BRMS should allow applications to be deployed in a service oriented architecture. The rule engine should therefore present itself as a service to applications and applications should be deployable themselves as services (e.g. as web services).

There are well known and understood technologies to support SOA. The use of these encourages, but does not guarantee, systems that supply services. SOA is not just about using the right tools and infrastructure, there is much more to it than that.

2.1.1 Business Drivers, Benefits and Pitfalls

The drivers for SOA are manifold. Perhaps the most striking is the history of failure of large IT projects referred to in Chapter 1, where we saw that maybe two-thirds of everything we do in IT goes awry. We need better project management. Here the news is good because the Standish surveys report steady improvement over the decade from 1994 to 2005. We need better requirements engineering too, but there is little convincing evidence of dramatic improvement here. In addition, evolving or poorly understood requirements point to the need to involve the business more closely in service definition and delivery. SOA is one way of moving closer to this goal. Evolving technology and platforms can also hit maintenance costs. Defining platform independent services through SOA should mitigate this tendency by providing a baseline of more stable definitions.

At the time of writing, the main focus of SOA adoption is 'behind the firewall'. Therefore, many of these considerations do not apply, but as the number of resources available on the web grows like Topsy, it becomes increasingly difficult to know if the service that you need is out there or not. SOA, as we have described it, makes it possible to search for available services and components much more readily. Finally, SOA is necessary for business-to-business transactions (B2B) on the web. In the short-to-medium term, of course, B2B links will be set up with great care and through negotiation by people – not by machines trawling cyberspace and matching parameters.

SOA proffers several significant benefits to its adopters. The ability to plug in new services without disrupting existing software, modularity based on business concepts rather than technical models, the ability to share and connect services across organizational units, companies and geographical areas. Most importantly, it can bring us closer to supporting business goals, especially when combined with a business rules approach, as we shall see. Adopting SOA, like business rules, encourages the separation of the concerns of business *versus* those of the infrastructure. With a well-designed SOA you can add value to your own business by doing things better and using other people's

services, and add value to other peoples' business by adding your services, thus sharing your savings with them.

Defining components as services also offers the possibility of contracting development out to third parties with better control over the results. If we add web services to this picture, the benefits of web delivery being well known by now, we can envisage immediate gains.

2.2 Service Implementation using Components

Component Based Development (CBD) is a sister technology to SOA; they are orthogonal, but sympathetic concepts; it is possible to have one without the other, but they best go hand-in-hand to produce component based systems that provide business services. For all practical purposes, service oriented architecture depends on component based development.

Component based design is aimed, like object-orientation, at improving productivity by offering a better chance of reuse through better modularity. If it is done well, we can build large, complex systems from relatively small components. If combined with an agile development process and if there has been sufficient investment in components, this can lead to a faster development cycle.

A software component is an object that is defined by an interface and a specification. An **object** is something with a name (identity) and responsibilities of three kinds: responsibilities for remembering values (attributes); responsibilities for carrying out actions (operations/methods); and responsibilities for enforcing rules concerning its attributes and operations (often referred to as constraints). An **interface** is a list of the services that an object offers. A **type** is such a list plus the rules that the object must obey (its **specification**). Contrast this with the notion of a class. A **class** is an interface with an implementation. A class has instances; a component has implementations. So what's the difference between a class and a component? Not much! Two things distinguish components. First, components are generally 'bigger' than classes; but this is not a distinction in principle. More importantly, components are delivered in the context of a framework within which they can interact: a 'component kit'.

Figure 2.2 shows part of the type specification of a queue component[3].

We can easily think of several ways to implement this type: as an array, as a linked list, etc. Sticking to arrays, there are still choices to make. We could use the implementation illustrated in Figure 2.3, where we maintain one pointer called 'last' and shuffle the values up to the front of the array and decrement the pointer by 1 when a 'leave' occurs. When there is a 'join' event, all we have to do is increase the 'last' pointer value by 1. Thus, $length = last + 1$.

[3]We are indebted to Alan Wills for this example.

Queue
length: integer
join leave
join increases length by 1 leave decreases length by 1

Figure 2-2 A specification.

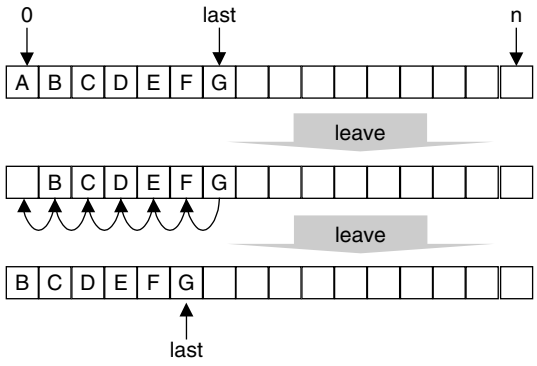

Figure 2-3 Implementation as an array with one pointer.

Alternatively, we can utilize the implementation shown in Figure 2.4, where we have to maintain two pointers but save some time on the shuffle operation. In this scenario, *length = last – first (mod n)*.

Here, *n* is the maximum queue length in both cases. The first element in the array is element 0.

So, we can interpret the queue as a component that can be implemented in multiple ways; but we could also interpret it as a component offering three services. A queue allows queuers to join and leave the queue and offers a service to retrieve the length of the queue at any time (except in the middle of a join or leave operation). Each implementation has its own algorithm for computing this retrieval of length, but each of these must satisfy the two rules in the specification. So, components look very much like service interfaces.

Thus, SOA can be viewed as a philosophy that drives the development of components by defining their interfaces clearly and in a way that relates to real needs. CBD then packages the services for development and maintenance, often using technology such as J2EE, .NET, CORBA and so on; although CBD is not necessarily about component middleware of this type. SOA and CBD both encourage the separation of specification and implementation.

Also, note that component specifications are almost always described using rules, as above. This is equally true of services. Rules of the type given above

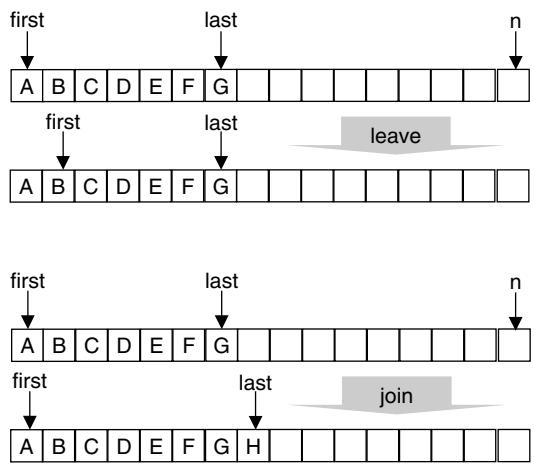

Figure 2-4 Implementation as an array with two pointers.

constrain the relationships among services; joining or leaving a queue changes its length.

Components, when delivered, must be documented and provided with a test harness. The documentation must state at least the following:

1. What the component expects of the environment into which it is placed: other components and services that must be present; their QoS parameters; etc.
2. Which services the component can be expected to provide: its specification.
3. Which business rules the component must conform to.

Another observation, coming both from work on component design methods and from the exigencies of current component frameworks, is that it is useful to classify components and organize the architectural layers around the classification.

Date (2000), Cheesman and Daniels (2000) and Andrews (2007) all identify a difference between general components and **core components**: objects or concepts that 'really' exist in the domain. Consider the potential components needed for building administrative systems for public libraries.

Clearly, Book is a core component, although we might profit from including a generalization of it such as Lendable_item. How about Member? No! Membership is a **rôle component**; the core object is Person. Membership is an association between people and libraries. Perhaps we should generalize the concept of library too, perhaps using Fowler's (1996) PARTY pattern. A library is a kind of organization which is, in turn, a kind of party. People are also parties. Another apparent core concept is the idea of a loan but, here too, there

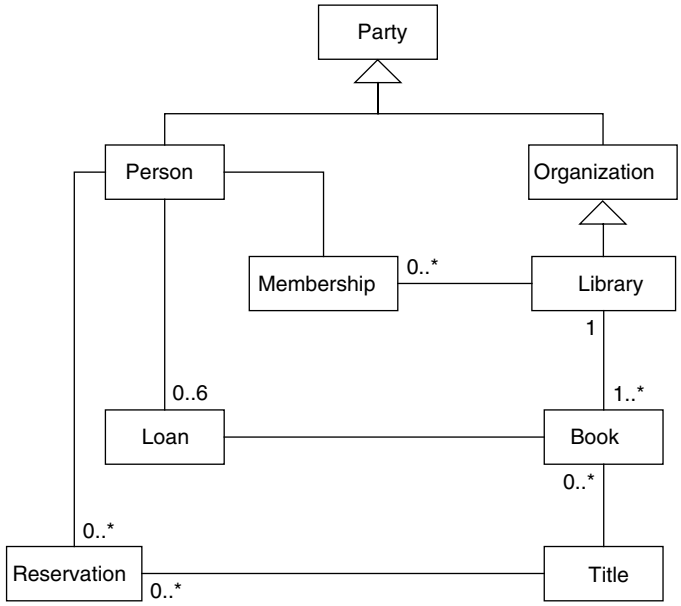

Figure 2-5 Some components for a library system.

are advantages in realizing that loans represent not things but a key business process of the library; a **process component**.

Figure 2.5 offers a UML type diagram representation of some of the components in a library. The cardinality constraints represent a particular kind of business rule. 1..* against Book is interpreted as the rule 'A library owns at least one and possibly many books'. 0..6 against Loan means that 'A person need have no loans but is not permitted more than 6 (at any one time)'. But there is another rule that is not shown on the diagram. We certainly do not want people who are not members to be able to borrow. This kind of rule is very common; whenever there is any kind of cycle in a UML class diagram then there is a potential rule lurking. Here are some of the rules that might be implicit in this diagram:

- The person borrowing a book must be a member of the library that owns the book.
- A reservation must be for a title that describes a book owned by the library.

Note that neither of these rules need be true (i.e. apply): interlibrary loans; purchase on reservation. So, perhaps the rules are more complex:

- The person borrowing a book must be a member of the library that is a member of the same interlibrary loans organization as at least one library that owns that owns a copy of that book.

- A reservation must be for a title has describes a book owned by the library unless there are four of more reservations for the same title, in which case the library will order one copy of that title.

There is a pattern here, our first in this book. The idea of patterns will be covered fully in Chapter 7, but bear with us whilst we present a thumbnail sketch of it.

Pattern 11	**ASSOCIATION LOOPS CONCEAL RULES**
Context	You are trying to DISCOVER BUSINESS RULES (5) and have completed part of BUILDING A TYPE MODEL (4). You know that you must WRITE THE CARDINALITY CONSTRAINTS AS RULES (12).
Problem	**How can you be sure that you have not missed any rules implicit in the type model?**
Example	Refer to Figure 2.4. Start with a person. Do they have a loan? If yes choose one. Every loan is for a unique book that has a unique title. Does the title have an outstanding reservation against it? If yes, go back to the person you started with. Does that person have a reservation? If so, is it for the same title? Perhaps the rule is: 'A member may not reserve a title which they have already borrowed a copy of'.
Solution	**Look for cycles (loops) in the type diagrams. Start at every type in the loop, choosing a generic instance of that type, and follow the associations to another type. Ask if each route brings you to the same instance. Write down the rule that says it does.**
Resultant context	The rules you have written down may not be true, so NOW ASK THE BUSINESS (13) and ASSIGN RULES TO COMPONENTS (14).

Words set in small capitals represent other patterns: patterns that we shall encounter later in this text.

So, working with component models reveals business rules. In the same way, service specifications reveal rules and are incomplete without them. Component models are a good way to round out the specification of services.

Component modelling is a crucial step on the road to good service oriented architecture. Modelling should encompass business matters and not just data and functional models of the software encouraged by many current UML tools and practices. As we shall see in Chapter 7, models must look beyond the system boundary and encompass all stakeholders, although models often

need to map to the platform architecture (.NET, J2EE, etc.) also. Models must always include business rules and constraints.

Business goals are supported by use cases. The type model is the vocabulary needed to express the goals of use cases. The pattern language of Chapter 7 begins with patterns that express this. We first ESTABLISH THE BUSINESS OBJECTIVES (1), then build a BUSINESS PROCESS MODEL (2) using use cases and focusing on their goals or postconditions rather than any notion of 'steps'. The use case goals **are** the services. As part of this we ESTABLISH THE USE CASES (3), DISCOVER BUSINESS RULES (5) and BUILD A TYPE MODEL (4).

Our requirements model needs to supply the software service specification:

- What business goals does it meet?
- What does it do?
- What information does it provide and require?
- What are the business rules?
- What are the quality requirements?
- What constraints and rules must be obeyed?
- Are there any interface dependencies?
- Who can use each service or component?
- What will it cost?

2.3 Agents and Rules

Agent technology has its roots in the study of distributed artificial intelligence, although the popularity of the approach has had to wait for more mundane applications in mobile computing, mail filtering and network search. Along with this plethora of new applications there is a great deal of very confusing terminology facing anyone attempting to understand the technology of intelligent agent computing. We read many conflicting and overlapping terms such as Intelligent Agents, Knowbots, Softbots, Taskbots and Wizards. Also, there are writings on network agents that are not true agents in the sense of most of the above terms. Furthermore, there are several competing definitions of an intelligent agent in the literature.

Russell and Norvig (1995) characterize an agent as 'anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors. A rational agent is one that does the right thing'. A report from Ovum (Guilfoyle and Warner, 1994) tightens this slightly: 'An agent is a self-contained software element responsible for executing part of a programmatic process, usually in a distributed environment.' Luck and McBurney (2005) say that agents are 'autonomous, problem-solving computational entities' that can operate in 'dynamic and open environments'.

From this viewpoint, agents are components that, rather than being invoked directly, can make choices among their permitted actions and interactions, as assigned by their designers and owners. An **intelligent** agent makes use of non-procedural process information – knowledge – defined in and accessed from a knowledge base, by means of inference mechanisms. But a more compelling definition comes from Genesereth and Ketchpel (Riecken, 1994): 'An entity is a software agent if and only if it communicates correctly with its peers by exchanging messages in an *agent communication language*.' This is a most important point if agents from different manufacturers are to meet and cooperate. Kendall *et al.* (1997) say that agents are objects 'that proactively carry out autonomous behaviour and cooperate with each other through negotiation', which further supports this view.

Agent communication languages (ACLs) perform a similar rôle to object request brokers or web services protocols like SOAP, and may be implemented on top of them. ACLs are necessary so that agents can be regarded as distributed components that need not know of each other's existence when created. There are two kinds of ACL, procedural ones such as General Magic's Telescript, and declarative languages such as the European Space Agency's KQML/KIF.

One might add that an agent is an entity that can sense, make decisions, act, communicate with other entities, relocate, maintain beliefs and learn. Not all agents will have all these features, but we should at least allow for them. One way to do this is to classify agents according to the level of features they exhibit; in order of increasing complexity and power these are:

- Basic software agents;
- Reactive intelligent agents;
- Deliberative intelligent agents, and
- Hybrid intelligent agents.

It is common to apply the description 'agent' to quite ordinary code modules that perform pre-defined tasks. This is an especially common usage in relation to macros attached to spreadsheets or database system triggers and stored procedures. Such 'agents' are usually standalone and have no learning capability, no adaptability, no social behaviour and a lack of explicit control. The term is also applied to simple email agents or web macros written in PERL or Tcl.

Reactive intelligent agents represent the simplest category of agent where the term is properly applied. These are data-driven programs; meaning that they react to stimuli and are not goal oriented. They perform pre-defined tasks but may perform symbolic reasoning, often being rule-based. They are sometimes able to communicate with other agents. They may have learning capability. At a macro level they may sometimes exhibit explicit control, but there is no explicit micro-level control. They cannot reason about organization. Homogeneous groupings of such agents are common. Examples of reactive intelligent agents

include monitor/alert agents encoded as a set of knowledge-sources or rules with a global control strategy. Service-oriented BRMS components often fall into this category

Deliberative intelligent agents are mainly goal-driven programs. They can have the ability to set and follow new goals. They typically use symbolic representation and reasoning; often using a production rule approach. They typically maintain a model of their beliefs about their environment and goal seeking status. They may be mobile and able to communicate and exchange data (or even goals) with other agents that they encounter. Deliberative agents may have learning capability. They can reason about organization and are able to perform complex reasoning. Their intelligence is programmed at the micro level at which there is explicit control. Heterogeneous grouping of these agents is possible. Data retrieval agents that will fetch and filter data from a database or the internet are typical examples of this kind of agent.

According to some authorities (Kendall *et al.,* 1997) this is the weakest permissible use of the term AGENT. **Weak agents** on this view are autonomous, mobile, reactive to events, able to influence their environments and able to interact with other agents. **Strong agents** have the additional properties of storing beliefs, goals and plans of action, learning and veracity, although there is some dispute over the meaning of the latter property.

Hybrid intelligent agents are a combination of deliberative and reactive agents. Such agents can be mobile and may try actively and dynamically to cooperate with other agents. If they can also learn, they are **strong hybrid intelligent agents**. Such agents usually contain (or may access) a knowledge base of rules and assertions (beliefs) and a plan library. An interpreter enables the agent to select a plan according to its current goals and state. When an event occurs a plan is selected (instantiated) to represent the agent's current intention.

It should now be easy to see that there are three prerequisites for agent computing: components, business rules and an ACL.

2.3.1 Agent Architecture

Adding rules to the interfaces of components has the useful side-effect of enabling us to model many intelligent agents and multi-agent systems without any special purpose agent-based modelling machinery. Agents are autonomous, flexible software objects that can respond to changes in their environment or context, engage in 'social' acts via a common agent communication language and be proactive in the manner of the Intellisense agents in MS Office, for example.

Intelligent agents are intelligent in the sense that they embody some kind of expertise or the ability to learn. This expertise may be encoded as business rules, in which case the agent must have access to an inference engine to process them. Learning algorithms are usually, of course, procedural in nature

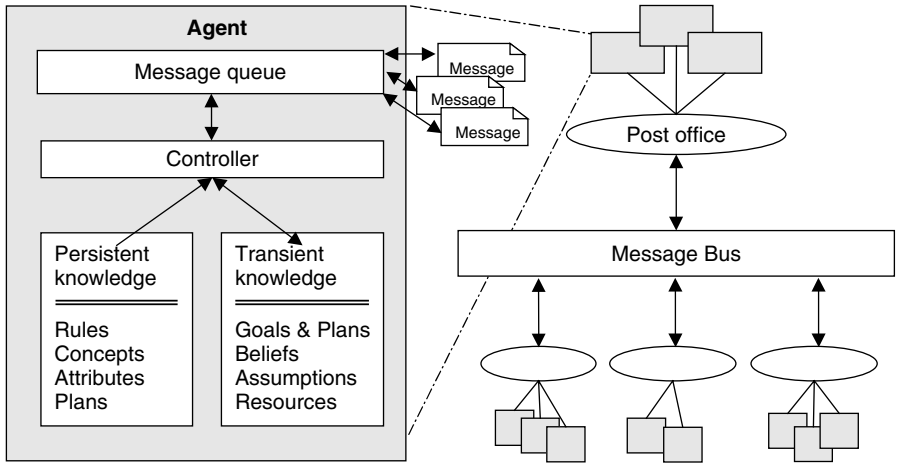

Figure 2-6 One possible architecture for an agent.

and may be based on decision branching (e.g. ID3), neural nets or genetic algorithms.

Agents need to communicate with users and with each other. As we have mentioned, the best way to do this is via a standard agent communication language. Unfortunately, such a universal standard is not yet agreed, so that designers are often forced to create one or work within a proprietary environment. The promise of SOA, especially in the light of web services, as we shall see, is that the ACL can be based on accepted industry standards such as SOAP and WSDL.

We can generalize about the basic architecture that most agent systems share. Figure 2.6 shows a typical architecture. Here, each agent has a controller that stores or can access an inference engine and problem solving strategy. The agent encapsulates two kinds of knowledge in its knowledge base: persistent and transient. The persistent knowledge often takes the form of attributes and methods that represent its ontology or type model, but the methods may be coded non-procedurally; Prolog perhaps or a BRMS rule language. The agent may also store knowledge about the rôles that it plays in the overall agent organization and about its plans. Plans, which are fixed, are to be distinguished from those that vary during execution, the latter being part of the agent's transient knowledgebase along with its current assumptions, beliefs, acquaintances and short-term goals. Agents communicate via a message queue and deliver messages to post offices on the network which, in turn, deliver to other agents, systems or users. Actual agent-based systems vary considerably but share this basic approach in outline at least.

One of the main reasons for the adoption of component technology is the move to distributed architectures. But *n*-tier architectures are often beset with severe network bandwidth problems. Using mobile agents can reduce the

amount of network traffic. The mobility of agents means that, when it is more efficient, we can send the program across the network rather than a request to retrieve unfiltered data. Smart agents can be used to personalize systems for individual needs and skills, which has the effect of reducing the cognitive and learning burden on these users. Agents that can learn, adapt and exchange goals and data can be used to speed up information searches, especially across networks or the internet.

The agent model is a model of distributed problem solving. There are several approaches to the co-ordination of distributed co-operating agents. These include centralized control, contracting models, hierarchical control via organizational units, multi-agent planning systems and negotiation models. These are not discussed further here, but one type of strategy is especially important for systems involving multiple, co-operating agents that each apply specialized knowledge to help solve a common problem. A common architecture for such applications is the 'implicit invocation' or blackboard architecture (Buschmann *et al.*, 1996). This becomes important for BRMS, as we will see in Chapter 7.

It should be apparent that adding rulesets to components is all that is necessary to make them into agents. Forward chaining (data-driven) rulesets enable reactive agents and backward chaining (goal-driven) régimes support deliberative agents. Because each object can contain more than one ruleset and each ruleset may have a specified régime, hybrid agents can also be built. Of course, learning abilities would not normally be represented as rulesets but, more likely, by operations – or a mixture of the two.

Repositories of business objects become, effectively, the domain ontology, extending the concept of a data dictionary, not only by including behaviour in the form of operations but by encapsulating business rules in an explicit form.

Intelligent agents may have to operate in the presence of uncertainty, and uncertainty comes in many guises: probability, possibility and many more. Modelling systems that restrict the logic in which rules or class invariants are expressed to standard predicate logic or first order predicate calculus (FOPC) are too restrictive. A better approach allows the designer to pick the logic used for reasoning: FOPC, temporal logic, fuzzy logic, deontic logic (the logic of obligation or duty), etc. Just as a ruleset has an inference régime, it has a logic. In fact the régime and the logic are intimately related. For example, standard fuzzy logic implies (usually) a one-shot forward chaining strategy that treats the rules as if they all fire in parallel. Unfortunately, current BRMSs offer scant support for uncertainty.

2.3.2 Applications of Agents

Agents have been used to monitor stock markets and trade shares automatically, to find and buy cheap flights and to collect data on a user's use of a

computer. They have evident uses in e-commerce where there are purchasing agents that can find the best price for a product, such as a book, across multiple web sites. An agent system handles malfunctions aboard the space shuttle. The White House uses e-mail agents to filter thousands of requests for information. MIT has built agents to schedule meetings. Sample applications of agent technology to date include data filtering and analysis, process monitoring and alarm generation, business process and workflow control, data/document retrieval and storage management, personal digital assistants, computer supported cooperative working, simulation modelling and gaming.

Agents in current systems perform information filtering, task automation, pattern recognition and completion, user modelling, decision making, information retrieval, and resource optimization based on negotiation (e.g. in air traffic control), routing.

Other current applications include:

- planning and optimization in supply-chain management;
- program trading;
- battlefield command and control simulation and training;
- network management and process monitoring (of networks and of business processes).

Another area where agent computing is becoming influential, at least as a modelling metaphor, is in business process modelling and re-engineering. The focus in most work on BPR is on process and this is as it should be. However, as Taylor (1995) has pointed out, an exclusive obsession with process can be dangerous because business depends on the management of resources and the structure of the organization as well as on effective processes. Thus, any approach to business process modelling needs to be able to model all three aspects: resources, organization and processes. It turns out that the agent metaphor when combined with an object-oriented perspective on systems analysis provides an effective solution to this modelling problem. Furthermore, modelling a user's responsibilities with an intelligent agent can often reduce the cognitive dissonance between the user's mental model of a system and its actual structure.

Components with rulesets support the modelling and design of intelligent agents and systems, but agents are also key to modelling business processes and reducing the cognitive dissonance between models of the world and system designs – an aim shared (we hope) by anyone implementing SOA.

Any agent worthy of the name provides a service. In the context of SOA, these services must relate to *business* services. But agents may also be consumers of services. In that sense, agent-based computing *extends* the ideas of SOA. Now let us look at some technologies that exist today and can be used to construct SOAs, and that may one day be the basis for a universally accepted standard for ACLs.

2.4 Service Oriented Architecture and Web Services

Web services provide a standardized way of interoperating amongst applications, regardless of the platforms they are running on. They realize SOA in a very practical way, using concrete agents that communicate by passing messages that conform to the standard protocols. The environment is open, in that agents can leave or join at will without disrupting the whole. Agents can act on behalf of service owners, to ensure contracts are met and relevant business rules enforced, or subscribers, to locate relevant services, negotiate contracts or deliver the results of service invocations.

Web services provide one possible infrastructure for SOA and indeed agent-based computing. Just as one can use SOAP without understanding SOA, one can adopt SOA without using SOAP, etc., but one cannot do SOA without good services.

Services are about using other (other peoples' and your own) systems as part of your own, these other systems offer functions, services, that you can use – your system will collaborate with these systems to achieve a goal. If disparate, distributed systems written in different programming languages are to communicate and collaborate with each other, they will need some sort of communication medium and a way of speaking to, and understanding, each other: a common language. A global communication protocol already exists: the internet, which is a mechanism for moving bits around. The first layer of abstraction built on top of this basic mechanism is for moving data around using TCP/IP. On top of this abstraction we can build web services; here the stuff that is moved around is XML and the mechanism used is intranet/extranet/internet. Building on top of this has many advantages; to use web services, there is no need to change the way that the infrastructure is used, web services represent just another application, and the existing internet protocols and infrastructure can easily be used; security applications, such as firewalls, will not become a problem.

Having got a pipe, some basic infrastructure, the web and XML, we need a language to describe the format of the messages and a mechanism to manage the interchange of messages. Since we have a distributed system, we also need to know where to look for things; these services are supplied by the following:

- WSDL (Web Services Description Language);
- SOAP (Simple Object Access Protocol);
- UDDI (Universal Description, Discovery and Integration).

A simple model will explain the functions of these various elements of web services. If we think back to before computers and the internet were central to communication, business could be done with letters carrying information

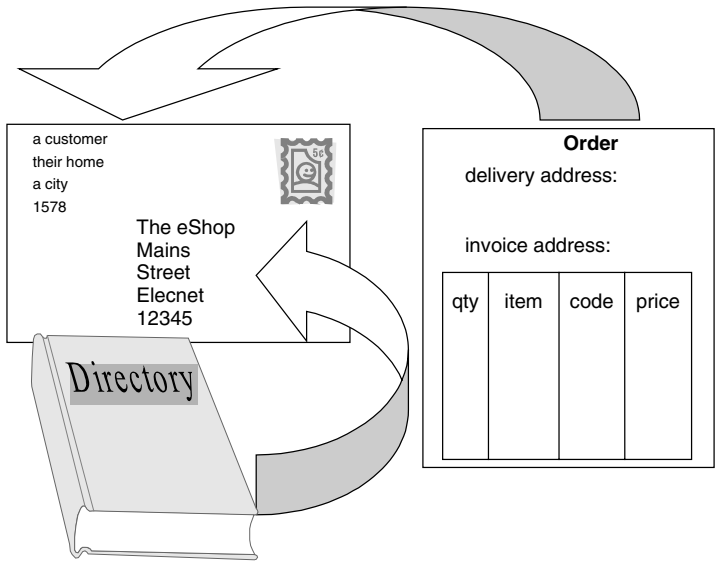

Figure 2-7 A simile of web postal services.

(the XML bit) and a PTT to deliver the letters (the web bit). To make business easier we might add some additional rules and services.

The postal service may define valid envelope sizes and where the stamp is placed, where the 'to' and 'from' addresses are written on the envelope, and where you post a letter – such standard will enable them to efficiently process your letter and speed its delivery to the correct address. To do business with another company, we can look up details of the company in either the yellow or white pages. Yellow pages classify businesses by type, white by their name.

To do business efficiently by post it would also be useful to have details of how to do business with the company. Information that describes what envelopes you should use, details of any forms you will need to complete (for example what needs to go on an order), any replies you might expect to get (the format of their invoice and delivery note) – an advanced postal service might publish these in a different coloured directory, say green pages. As a user using the directories, you fill out the necessary form or write the appropriate letter, put it in the envelope and post it to the address found in the directory and possibly await a reply if the rules tell you to expect one. The simile is illustrated by Figure 2.7.

UDDI is about the rules of the business including its whereabouts and how you contact it and what information needs to be exchanged to do business. WSDL is about the information you need to supply using various forms and letter formats in an exchange with the business; this is described in the business directory green pages. SOAP is about defining the rules of the post office. SOAP is a very simple postal service, if you need a more services such as recorded delivery, proof of posting, tracking of your letter, insurance and

receipt of delivery you need more rules; this is what ebXML or BIZTALK are about – but these are not part of this discussion.

XML

In addition, we may well use XML, the acronym for eXtensible Markup Language, which is designed to structure data by marking individual items with tags that hint at what each item represents. The format of a particular XML document is described by a Document Type Definition (DTD) or in an XML schema (leading to the claim that XML is designed for defining XML documents). Thus an XML document with its DTD or XML Schema is designed to be self-descriptive.

XML is a language for communicating instances of abstract syntax (and for defining those abstract syntaxes). The concept of abstract syntax has been around for nearly 40 years; this is where the information being captured is separated from the concrete syntax used to write the information down. Of course, we need to write the abstract syntax down, but it is at a higher level of abstraction – we represent the information as a tree. XML is a flattened representation of a tree. Consider a simple example, in Figure 2.8, representing an if-then-else statement using abstract syntax.

This description will work for any language that has a traditional if-then-else statement; the keywords are left out; but we still have a representation of the information content of the statement. In XML we could represent an if-then-else statement, thus:

```
<ifThenElse>
  <BooleanExpression>
    ...
  </BooleanExpression>
  <then>
     <statement> ... </statement>
    <statement> ... </statement>
    ...
  </then>
  <else>
     ...
  </else>
</ifThenElse>
```

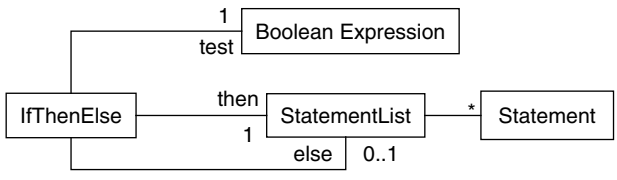

Figure 2-8 If-then-else as a UML type diagram.

It can be seen that XML can be used to encode objects, attributes and links, to flatten a collection of objects, their attributes and links into a form suitable for sending around the internet. Using XML, we can pass data around the internet as character strings that encode objects; and with SOAP we can call methods in distributed objects passing the structured data as arguments.

Note that an XML document does not do anything, it is just information marked with XML tags; do not run away with the idea that there are any semantics embedded in this information – that is encapsulated in the software that sends or receives the XML document; the software will have an understanding of what the information is about. The following example is a name and an address written using XML:

```
<address>
  <person>
    <firstname>Derek</firstname>
    <surname>Andrews</surname>
  </person>
  <no>356</no>
  <street>Main Road</street>
  <town>Glen Magna</town>
  <county>Leicestershire</county>
  <postalcode>LE34 7NH</postalcode>
</address>
```

An address has information about a person and where they live. An address has the house number, street, town and county – a fairly standard layout for an address in the UK, and fairly self-explanatory: it is self-descriptive. However, there are limitations. If you believe that this XML document does have some semantics, consider the dilemma faced by someone asked to supply address information to an American company. They may be asked to supply the following data:

```
<address>
<person>
 <givenname>Derek</givenname>
 <familyname>Andrews</familyname>
</person>
<buildingno>356</buildingno>
<street>Main Road</street>
<city>??Leicester??</city>
<state>??England??</state>
<zip><state code>LE</statecode>
<zipnumber>??34 7NH??</zipnumber>
</zip>
</address>
```

We have little problem with the person part, but is the city 'Leicester', the postal city for 'Glen Magna'? What do I supply for the state? Is it

'Leicestershire', but that is a county, and I know there are counties in the USA. . .Is it 'The East Midlands', the area I live in, or is it 'England' which could be considered a rough equivalent to state as it is the next thing up from a county, and the state of the UK is made up from the countries of England, Wales, Scotland and the province (not a country) of Northern Island. There are even more problems with the zip code; I will assume that I can use LE as the equivalent of the state code, but the rest of the post code is not numeric, and is meaningless as far as the UK encoding goes. The American company and I do not agree on the interpretation of the XML tags. It is not enough just to invent tags and hope that their names provide a meaning, the semantics must be given as well.

A subtler example is the following information interchanged by two hotels in a hotel chain about room occupancy:

```
<roomdetails>
  <roomno>34</roomno>
  <roomtype>double</roomtype>
  <occupant>Jo Smith</occupant>
</roomdetails>
```

Further investigation reveals that the message format only allows a room to have only one occupant no matter how many people can sleep in it – in this example we have a double room. What is the explanation? You need to know that the hotel chain doesn't care how many people are staying in a room; it only wants to know who is responsible for paying for it, the occupant. Providing both the sender and the receiver of this message are aware of this, there is no problem, but this information cannot be deduced from the XML alone, you need to know about the business and its rules.

XML can be used to exchange data between two incompatible systems. In this case it will be used a standard format that both ends of the transaction can understand and can translate their representation of the data to and from. In fact a message standard can be expressed in XML for many systems to exchange data, though they all will need to agree on the tags and the meaning of the data associated with those tags. As XML is character based, there is an additional advantage that one of the machines that can understand the data is the human brain. One of the advantages of XML is that we can extend a message format by adding additional tags to mark the additional information carried in the message. Existing applications will ignore the additional information; new applications can recognize and use it. Since the information is encoded using text, with XML, plain text files can be used to store data. XML can also be used to store data in a database. Applications can be written to store and retrieve information from either text files or database, and generic applications can be used to display the data – for example the XML can be translated into XHTML and displayed using a browser.

When deciding how information is to be encoded with XML, users of that particular piece of XML must agree on the names of the tags, their structuring (nesting) and most important the meaning of the data marked with a particular tag. From the above example it should be noted that the name of the tag is not enough for this, though it does help. Any one taking part in the interchange must have a shared (possibly dynamic) vocabulary, which includes the meaning of any terms (tags) used in a conversation.

SOAP

Middleware software such as CORBA, .NET and J2EE supply some sort of Remote Procedure Call mechanism (RPC), but it is not secure and there are compatibility problems communicating between different programming languages and different middleware, each must be supported by the mechanism. There is a requirement for a mechanism that avoids these problems: one based on HTTP would work since this is supported with browsers and servers and by the internet. SOAP is a communication protocol for exchanging information between applications; it wraps up a document and moves it over the internet. A SOAP document is encoded using XML. A SOAP method call is a HTTP request-response that conforms to the SOAP rules; think of it as being the procedure call mechanism for a service that uses a WSDL definition that gives the name of the service and describes any parameters that are needed.

A SOAP message looks approximately like this, leaving out some of the more complex parts.

```
  Information to help with understanding and routing the
   request to the server:
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-
encoding">
<soap:Header>
...
...
</soap:Header>
<soap:Body>
  ...
  ...
  <soap:Fault>
  ...
  ...
  </soap:Fault>
</soap:Body>
</soap:Envelope>
```

If the Header element is present, it is the first part of the Envelope. This header contains application specific information which is about any bureaucracy surrounding the service such as details of the transaction it belongs to where two or more services need to be processed together (or not at all). Other services such as authentication, encryption used, any payments that are due and any other additional information that may be needed can also be placed here. This allows additional information to be added over time without breaking the original specification. The Body element is required and contains the actual SOAP message intended for the ultimate destination; it contains the name of the procedure and the arguments of the procedure call in programming terms. An error message is carried inside the optional Fault element. It is the contents of these three fields that are described in the UDDI using WSDL. It should be noted that because of the way SOAP is designed, its use is not restricted to use over the internet, it can also be used over other transport mechanisms such as email and message queues.

WSDL

WSDL is the acronym for Web Services Description Language. WSDL is an XML-based language which is used to describe a web service's capabilities by providing information about the business including its internet address, services provided and the message formats. In programming terms WSDL is the procedure declaration and SOAP is the procedure call mechanism. WSDL is an integral part of UDDI, an XML-based worldwide business registry. WSDL is the definition of the procedure call parameter names and types, but more general and expressed in XML; structured data can be used as parameters, rather than just simple data values.

Web services can use business services, but not necessarily the other way around; they are at different levels of abstraction. Just because you expose an API to the world as a set of web services, it does not mean you have SOA.

The format of a WSDL document follows:

```
<definitions>
<types>          Definitions of the data types that will be used in the
                   messages – these are machine- and programming
                   language-independent.
...
</types>
<message>        Definitions of the messages that will be transmitted; these can
...                be parameters with input separated from output or document
                   descriptions (parameters in a procedure declaration).
</message>
```

`<portType>`	*What operations and function will be performed by the web service; these will refer to message definitions in the<message>section to describe function signatures: the operation name, input parameters, and output parameters. (C.f. the procedure name and parameters – the WSDL equivalent of a Java interface.)*
`...`	
`</portType>`	
`<binding>`	*Specifies binding(s) of each operation in the portType section, describes how the messages will be transmitted – the communication protocols to be used by the web service and further information about the operations defined in portType – these will be specific to the underlying web protocol used for exchanging the SOAP messages (there are three recommended protocols: HTTP, HTTP GET/POST, SOAP/MIME).*
`...`	
`</binding>`	
`</definitions>`	

UDDI

UDDI is short for Universal Description, Discovery and Integration. It is an XML-based directory that enables businesses to list themselves on the internet so they can be found by other businesses. A UDDI entry for a business providing web services consists of three main components that define what the businesses are, where they can be found on the internet and how the businesses can interact with each other over the internet. It is the web equivalent of a telephone directory with both yellow and white pages, together with additional information store in 'green' pages.

An entry in the white pages provides the basic contact information about a company, such as the business name, address and contact information. These may also provide a unique business identifier, such as Dun & Bradstreet's D-U-N-S number; these are unique nine-digit sequences for uniquely identifying a business. The white pages allow customers and business partners to discover business services based upon the business name.

An entry in the yellow pages describes the business services using different categories (e.g. being in the manufacturing or the software development business, as per a yellow pages telephone directory). This information allows others to discover business services based upon its category. For example, a service might be categorized as an 'Online Store' service and at the same time be categorized as a 'Book Store' service.

An entry in the green pages provides technical information on the behaviour and use of the business services that are offered, and any support functions supplied. Green pages in UDDI are not limited to describing XML-based Web services used over the internet, but any (electronic) business service offered

by a business. This includes phone-based services such as call-centres, E-mail based services such as technical support for a product, fax-based services such as a fax to E-mail service, etc. Information such as the service location, the category to which this service belongs, and the specification for the services can all be found in the green pages.

A UDDI directory is designed to be interrogated by SOAP messages to provide access to WSDL documents that describe the protocol bindings and message formats required to use listed web services. These descriptions are encoded using XML. It should be noted that UDDI does not necessarily have its services described in WSDL for use by a SOAP call, other protocols can be used – for example a fax service would have described the protocol used by the fax, and email the type of messages supported (plain text or HTML for example).

Of the few companies that have pioneered service oriented architecture, fewer still have succeeded. Often they have proceeded by wrapping their existing systems so that they present themselves in the form of lots of web services. They end up with a business process (or application) layer that makes hundreds of very low level calls through the bus to the web service interfaces of the legacy systems. These point-to-point connections give immediate payback and get the job done, but at the expense of the creation of a vast amount of 'spaghetti' – far, far worse than when the same was accomplished by running wires between boxes.

This approach leads to:

1. Thousands of low level interfaces;
2. Brittle topology;
3. Poor extensibility;
4. Poor understandability and maintainability;
5. No reuse; and
6. Absolutely none of the promised benefits of SOA.

What is needed is an abstraction layer of business services (typically realized by components) that sit in the no-man's land between business and IT – and are understandable to both. These can then be reused and extended in line with changing business goals and priorities.

These services must be specified rigorously, not just in terms of XML schemata but also in terms of their process behaviour.

A possible acid test for SOA might be this. Give an executive director a pen and the back of an envelope, and ask her to draw boxes to represent the major services she uses to deliver the organization's business goals, to say which boxes are services provided by people and which by IT, and to describe how the services talk to each other – all at the highest level. We suspect that few companies are at that level of SOA maturity.

If SOA is limited to bottom-up, convenience grouping of low level API services, understood only within the IT department, business people will understand none of it; nor will the IT dept be any clearer on overall business goals and strategy.

SOA, done properly, will both provide and be based on a common language for business and IT. Then, existing and future systems can be discussed readily – using the names of the business services – the names of the boxes on that envelope. Service orchestration can then be done in business terms at the highest levels: I want this login service, this stockcheck service, etc.

2.5 Adoption Strategies

How does one build a world class rule-based, service oriented architecture? Moving from your current approach to software development to one geared towards an SOA approach will involve some changes. Building a business model is too often avoided as it is seen as extra, non-productive work. RUP (Kruchten, 1999) suggests writing lots of descriptive use cases as part of the requirements process. Both of these attitudes need to be changed. To find services, you need to understand the business, and a quick way to understand the business is to build a business model. Use cases are usually interpreted as being about developing the user interface, and the real user of the system is frequently not the person in front of the keyboard and screen. Thus we are interested in finding the real users of the system and their goals; then we can supply the services that will help them achieve those goals.

Moving from a business model to a working CBD/SOA system is a new skill to be developed, and using existing resources in a SOA involves careful development work. An architecture has to be designed and developed, and modified as experience is gained. Higher quality will need to be built into the software, as it may be seen outside the business or originating department. All these considerations will affect the way systems are developed and maintained. Our approach is broadly as follows.

As a first step, build high-level, abstract models of the business goals, business processes and business entities and concepts with the aim of:

- integrating different parts of the business;
- identifying reusable components that provide services;
- identifying the business rules that must be obeyed by these components;
- identifying common services and specifying them; and
- identifying the reuse of legacy systems.

Think about re-engineering the business, contracting out some services, contracting in others and especially doing new things by using other peoples' services. The slogan should be 'stop doing old, unprofitable things'.

Next, turning to the question of architecture, try to match your technology to the services defined. Then focus on business processes. Look for easier and new ways of doing business – reduce the number of business rules. Adopt agile development processes. Standard processes are not adequate. You will need additional or different tasks in the process, a different type of specification.

Adopting a BRMS will assist in the transition to SOA because service-based and legacy applications can be coupled using the BRMS as the common decision engine. In the transition period, decision logic is gradually extracted from the legacy and replaced with calls to the rule service (together with some code to interpret the responses). In this way, the legacy can be incrementally replaced by services. Most major BRMS vendors provide wizards for setting up a rule service using web services.

2.5.1 After SOA

'We have SOA, what are the benefits we should be seeing?' SOA, if done properly, should lead to more efficient business process, better process modularity and even business process reuse, but you must focus on the business and not the software.

'We have adopted SOA and have developed one or two successful systems; now what?' Are there any additional advantages? Since the services are about the business, and the business rules have been isolated in an appropriate component, it is easier to change the way the business works. Services supplied by other businesses can be absorbed into our own, if appropriate, and services we supply can be given to other businesses. Business rules can be simplified, and the way the business does its work can be changed for the better. Building SOA based systems encourages an understanding of the business and the way it works, this can lead to changing the business.

Existing legacy systems can be analysed and wrapped to supply services and, using some well-understood techniques, evolved into the new business structure. There are also techniques for replacing legacy systems over long periods so as to not impact the business.

As the emphasis is on modelling the business, by building a business (analysis, requirements) model (a Computation Independent Model using OMG terminology), the development team are in a position to investigate and exploit MDA, leading eventually to even cheaper and faster software development.

We can illustrate the importance of getting the interface right and the importance of emphasizing *business* services with the following example. Consider the problem of refuelling an airliner between flights at an airport. A clerk will use information about the flight (number of passengers, destination, cargo, plane type etc.) to calculate the optimal fuel load and send an order to the fuel company to refuel the plane with the necessary amount of aviation fuel when the plane arrived. Even if the order is placed electronically, it is likely

to be placed some time before the refuelling is actually needed as the airline company has no idea of the availability of the fuel trucks; this is the problem of the oil company. In order for this to happen, the airline company will need to use estimates of the aircraft load from passenger and cargo bookings. Their requirement is to know fuel requirements ahead of time so they can schedule the trucks and necessary staff. There are two problems here: the software will be written for the clerk to use to work out the necessary fuel needs (the clerk may be automated and send an email order, but this does not change the problem). However, the real user of the system is not the clerk, it is the guy refuelling the plane, he needs to know how much fuel to load, this is what this part of the airline business is about. A service oriented interface will be about loading the correct amount of fuel onto the plane, nothing else; it is not about a conversation with the clerk as to airline type, destination, load, etc.

The message here is that you need to get outside of the system boundary to identify services. The system boundary tells you the services the interface designer wants, not the real user (who may not be the person using the computer – think call centre, the real user is the customer on the end of the telephone, not the call centre clerk!).

By writing detailed use cases we are concentrating on the user interface rather than what is happening in the real world, we are focussing on the solution rather than on the problem. Frequently this traditional approach leads to lots of documentation of the use cases, and little understanding of the business; detailed use cases are about the design of the user interface – this is an activity that we can leave until later. There is a need to do the description at a level that provides both understanding and provides a basis for detailed development of the HCI later. We need to move our perception up a level – at an abstract level we are interested in the goals of the business so we can supply business services that help in achieving these calls. The real user will need to be supported with services; the system user is supported by a user interface – low level stepwise use cases are about defining the system user interfaces.

Use cases define the interaction of the actors with the system, but the real user of the system may be far outside the system boundary. For example, in a library example: the library users want a reservation to be fulfilled when it is their turn in the queue of members waiting for a particular book. When they return a book, they want the transaction recorded so they are no longer responsible for the book they were loaned. Our library members are not that interested in the actual loan being recorded – though the library is very interested in that particular transaction. Thus the purpose of the system is to make certain that the business of the library is run properly. We should be interested in the goals of the real-user who can vary between the customer and the business, the goal of any actors using the system are not usually relevant for finding system services. There is a further advantage,

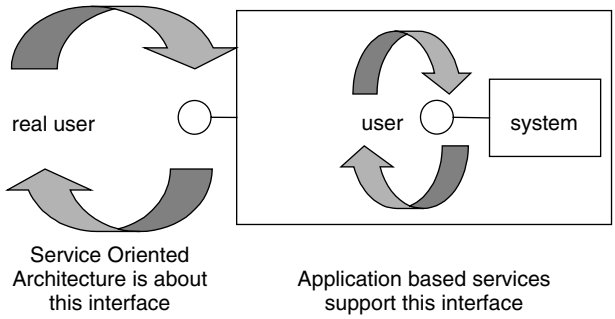

Figure 2-9 The provenance of services.

if we write the system with services in mind we can consider all sorts of interesting changes. Sometimes the real users are actors – customers buying goods online – and it is still better to find their goals, not the system's. As suggested by Figure 2.9, there needs to be a change of emphasis – we need to understand the needs (goals) of the real user and supply services that help to satisfy those needs.

The way to identify business services is not by analysing the use of the system by an actor, but by analysing the business. As Graham (2001) points out, a transaction in a business is usually carried out as a conversation between two or more parties who are interested in some sort of joint goal – making a sale, obtaining a book on loan. It is the goal of the conversation that is of interest to us, this is the service and it is this that we will supply machine help with.

With a service oriented approach, the move from the real user using the system through an intermediary to using the system directly should be straightforward; just changing the user interface should be the maximum amount of work that needs to be done. More exciting changes can be made with a service approach, there is a possibility of changing who does what. Suppose you request services from a supplier. For example, returning to the problem of refuelling an aeroplane at an airport. The current approach is likely to be ordering the requisite amount of fuel from the fuel company. An extreme approach would be to get the oil company to fill the plane as necessary with a bonus of the oil company sharing any cost saving with the airline. This could lead to just in time refuelling, where the optimal amount of fuel is loaded on the plane by the oil company, and the airline managing the optimum cargo/passenger mix. In a more general case, we move from a client/supplier approach to a partnership approach or even a consumer/supplier approach with the supplier driving the business with the incentive of profit sharing. The migration from one approach to the other is easily managed if we have services which are about the business rather than services that are about the interfaces. We can also see with the migration of control that there are cost savings that

can be made, and these can be shared with partners as an incentive to move in this direction.

Generalizing, we are moving along the following delegation of control: from the client being in control to the service provider being in control, with these intermediate possibilities:

1. Do exactly this for me.
2. Will you do this for me? Here is the information you will need to work out what I want.
3. Will you do this for me? Here is access to the information you will need to work out what I want.
4. Do this for me when I need it – I will tell you when. Ask for any information you need to work out what I want.
5. Do this for me when I need it; ask for any information you need to work out what I want and when I want it.

If the original system was set up correctly to supply services, this migration of control from the client to the service provider is reasonably straightforward. The service provider can move to a just in time service with some negotiation with the client about fuel *versus* plane load.

One way of testing for SOA at a global scale has already been given, on a smaller scale we can look at the messages that pass between actors to confirm a conversation, these are the messages we need to model in our system, since one of the participants in a conversation is likely to be replace by a machine.

If you have a paper system with clear understanding of the information flowing around, it is much easier to re-organize things to improve productivity or to take advantage of new business opportunities. The same principle holds for computer based systems.

In a nutshell, low coupling and high distribution give the service provider control; high coupling and low distribution (SOA) give the client control. Ultimately, if SOA is implemented well, the real user may be willing to do more work; and share the cost saving with the IT function.

2.6 Summary

We looked at the nature of SOA and its connexions with BRMS. They have in common the aim of raising the level of abstraction closer to the concerns of real business users; the real user not the IT department or administrative users; systems for the business, for employees, for customers or for suppliers; systems for the real user not the 'user'. SOA provides services that help people perform tasks that deliver them value. BRMS goes further and separates the rules from the code. Without BRMS, SOA is less effective and harder to maintain. SOA starts with business objectives and processes and focuses on a reusable service

level abstraction layer. Rules are parts of its specification. Components are grouped into services during design.

We saw that CBD was a very natural way to design for SOA and met some analysis patterns that are relevant to SOA, CBD and BRMS. SOA, CBD and BRMS are complementary technologies. We also saw that rule based components could be regarded as intelligent agents.

SOA is not the same as web services but the latter is one way to implement it. Beware of low level calls in the middleware, leading to a spaghetti of low-level point-to-point connexions. Don't start with wrappers; don't start with technology.

Let us put all this together. SOA, BRMS and CBD, used together offer potentially tremendous business benefits. In future we may see the technology broaden out into intelligent agent architectures that unite and enrich all these technologies. Web services have an important rôle to play as an implementation technology and may provide the basis for future ACLs.

Lastly, we considered the issues facing adopters of SOA.

2.7 Bibliographical Notes

The fundamental ideas of component-based development were described, from different points of view and emphasis, by Sims (1994), Szyperski (1998), D'Souza and Wills (1999), Cheesman and Daniels (2001) and Graham (2001). Andrews' forthcoming book (2007) focusses on the design of component-based systems and service oriented architectures and, in particular, extends the ideas of John Daniels and Alan Wills in the form of the Catalysis™ II method.

There is a large literature on agent-based computing and intelligent agents. Notable works, from our present point of view, include Farhoodi (1994), Ferber (1995) and Graham (1998; 2001).

The internet is a source for many discussions on and (competing) definitions of service oriented architecture.