Requirements Modelling and Specification for Service Oriented Architecture

Ian Graham

WILEY
A John Wiley and Sons, Ltd., Publication

Requirements Modelling and Specification for Service Oriented Architecture

Requirements Modelling and Specification for Service Oriented Architecture

Ian Graham

WILEY
A John Wiley and Sons, Ltd., Publication

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wileyeurope.com or www.wiley.com

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Ltd, 6045 Freemont Blvd, Mississauga, Ontario L5R 4J3, Canada

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

Graham, Ian, 1948-
Requirements modelling and specification for service oriented architecture / Ian Graham.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-77563-9 (pbk. : alk. paper) 1. Web services. 2. Software architecture.
3. Computer network architectures. 4. Business enterprises – Computer networks.
5. Computer software – Specifications. I. Title.
TK5105.88813.G73 2008
006.7′6 – dc22

2008031767

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 978-0-4707-7563-9

Typeset in 11/13 Palatino by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Bell & Bain, Glasgow
This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Contents

Foreword by Mark McGregor **ix**
Foreword by Professor Neil Maiden **xi**
Preface **xiii**

1 Principles of SOA **1**
1.1 Why Projects Fail 1
1.2 Aligning IT with Business – Speaking a Common Language 3
1.2.1 Models 6
1.3 What is Service Oriented Architecture? 8
1.3.1 The Real User 16
1.4 Business Drivers for SOA 19
1.5 Technology Drivers 20
1.6 Benefits, Pitfalls and Prospects 23
1.6.1 Pitfalls 24
1.6.2 Post-SOA Benefits 25
1.7 Migration Strategies 26
1.8 Summary 27
1.9 Bibliographical Notes 30

2 Architecture – Objects, Components, Services **31**
2.1 What is Architecture? 31
2.1.1 Architecture as High Level Structure 32
2.1.2 Architecture as Design Rationale or Vision 37
2.1.3 Architecture and Reuse 41
2.2 Architecture through the Ages 42
2.3 Objects and Components 49
2.3.1 Components for Flexibility 53
2.3.2 Large-Scale Connectors 54
2.3.3 How Services Relate to Components 56
2.4 Architecture and SOA 57
2.5 Stateless Services 63

2.6 Practical Principles for Developing, Maintaining and Exploiting SOA 66
2.7 Summary 68
2.8 Bibliographical Notes 70

3 Approaches to Requirements Engineering 71
3.1 Conventional Approaches 71
3.1.1 Approaches Based on Human Factors 73
3.2 Classic Requirements versus Use Cases 78
3.2.1 UML Basics 78
3.2.2 Use Case Models 80
3.2.3 Formulating Requirements 83
3.3 Problem Frames 85
3.4 Requirements and Business Rules 88
3.5 Establishing and Prioritizing the Business Objectives 89
3.6 Soft Techniques for Requirements Elicitation 93
3.6.1 Using Interviewing Techniques 93
3.6.2 Repertory Grids 96
3.6.3 Hierarchical Task Analysis 97
3.6.4 Object Discovery Techniques 101
3.7 Summary 106
3.8 Bibliographical Notes 110

4 Business Process Modelling 111
4.1 The Origins of and Need for Business Process Modelling 111
4.2 Business Process Modelling in a Nutshell 114
4.3 UML Activity Diagrams 116
4.4 BPMN 118
4.4.1 Fundamental Business Process Modelling Patterns 121
4.4.2 A Practical Example 124
4.5 WS-BPEL 127
4.6 Orchestration and Choreography 129
4.7 Process Algebra and Petri Nets 130
4.8 The Human Side of Business Process Management 135
4.9 Summary 136
4.10 Bibliographical Notes 136

5 Catalysis Conversation Analysis 139
5.1 What is a Business Process? 139
5.2 Conversations 141
5.3 Conversation Stereotypes and Scripts 145
5.3.1 Handling Exceptions 147
5.4 Conversations as Components 149
5.5 Contracts and Goals 151
5.6 Conversations, Collaborations and Services 155
5.7 Checking Model Consistency 160
5.8 Summary 161
5.9 Bibliographical Notes 163

6 Models of Large Enterprises 165
6.1 Business Process Modelling and SOA in the Large 165
6.2 Business Rules in the Mission Grid 173
6.3 The Mission Grid as a Roadmap for SOA 176
6.4 Other Approaches 177
6.5 Summary 177
6.6 Bibliographical Notes 178

7 Specification Modelling 181
7.1 From Requirements to Specification 181
7.2 Some Problems with the Conventional Approach to Use Cases 182
7.2.1 Overemphasis on Functional Decomposition 183
7.2.2 Lack of Clear Definition 183
7.2.3 Controller Objects 184
7.2.4 Use Cases and Scenarios 184
7.2.5 Essential or Generic Use Cases 185
7.2.6 Atomicity 186
7.2.7 Level of Abstraction 186
7.2.8 Exception Handling 187
7.3 Describing Boundary Conversations or Use Cases 189
7.4 Establishing the Type Model 192
7.4.1 State Models 193
7.5 Finding Services from State Models 198
7.5.1 Cartooning Using Agents or Co-ordinators 199
7.6 Finding Business Rules 201
7.7 Ontology, Type Models and Business Rules 207
7.7.1 Rules and Rule Chaining 208
7.8 Documenting the Specification 212
7.9 Associations, Rules and Encapsulation 212
7.9.1 Integrity Rul*e*s, Rulesets and Encapsulation 216
7.10 Summary 218
7.11 Bibliographical Notes 220

8 Standards 221
8.1 BPM Standards 221
8.2 Web Services Standards 224
8.3 Other Miscellaneous Standards 224
8.4 Bibliographical Notes 228

Appendix A Requirements Engineering and Specification Patterns 229
Appendix B The Fundamental Concepts of Service Oriented Architecture 271

References and Bibliography 281

Index 289

Trademark Notice

Foreword by Mark McGregor

Author of Thrive – How to Succeed in the Age Of the Customer and In Search of BPM Excellence; former Chief Coach BPMG.org

When I was asked to write a foreword to this book, I was a little surprised. I have known Ian for many years, and he knows me pretty well. 'Why?', I thought, 'would someone with such a strong technology and methodological background ask me, well-known industry cynic, to do such a thing'.

Then I read the book and all started to become much clearer; this really is the first book of its kind that I have read. Instead of trying to squeeze round pegs into square holes, Ian has taken the time to remind us of which pegs go where and then proceeded to provide us with clear guides on how to make it work.

Few could argue that good IT systems aren't critical to the success of almost any business today. Historically, many of the IT problems we see have been blamed on a disconnect between the business users and IT people. In many ways this disconnect has been propagated through books and articles telling us what and how SOA and Business Process Management (BPM) are, and how we should use them.

As someone who has spent much of the last ten years operating in the BPM and modelling sectors, the problem I see is that too much has been written about how BPM and SOA are about technology and, on the other hand, too much about SOA as a purely business problem. Such confusion inevitably slows down, rather than speeds up, technology adoption.

So at last it has happened. Ian Graham has taken a long look and reminded us that if we are to try and solve the right problems, then first we have to put them in the right boxes. More than that, in this book he clearly puts forward logical arguments and gives great examples of how by doing this we can deliver better business results and systems in a faster and more cohesive manner.

Unlike many who write in this area, Ian has experience behind him; SOA is talking to a core part of development that he has spent many years working in, not just as an author, but as a practitioner – speak to him

privately and he will show you the scars. For many, SOA is all shiny and new, but to old hands it is just the next logical evolution of the paradigm that started when software development started down the object oriented route. Object Oriented, Component Based, Web Services and now SOA! Ian has been instrumental in pushing the boundaries and helping people leverage these for success right from the very beginning.

For me, this book really works well in putting SOA into the correct context. It reminds us that sometimes business processes are just the words that IT use to describe systems processes, and when they use the phrase they are very often really talking about system requirements capture or analysis.

With the clear step-by-step methodological approach proposed here, any organization that is considering putting SOA in place will find themselves with a head start compared to those who are still mixing up the terms and ideas needed for IT supported business innovation.

I am certain that all readers of this book will finish it with a greater understanding of what to do and have a whole host of ideas that they can readily apply. Thank you, Ian

Mark McGregor

Foreword by Professor Neil Maiden

City University, London

Ian Graham's new book is most timely and needed. Recent developments in service-centric computing have been rapid, with worldwide spending on web services-based software projects reaching record levels, and more and more projects exploring web service technologies.

These developments are already having a major impact on how to specify and develop service-based systems. However, most projects still lack effective processes, methods and techniques to develop these systems. Fortunately Ian Graham's book provides some solutions. He builds on his previous experiences in requirements engineering and component-based development to link requirement and design practices with web services for the first time.

This book will teach you about web services and their evolution from software components. It will inform you about web service approaches and standards such as BPEL, business process modelling and business process engineering, and topics increasing important to requirements practitioners and web service developers such as ontologies and business rules.

I strongly recommend this book to anyone wanting to understand how to adapt their software development processes to implement service-based applications.

Professor Neil Maiden

Preface

Service oriented architecture (SOA) is becoming the modern way of conceptualizing software engineering best practice. This book covers the early stages of SOA projects and shows how to specify services based on a sound understanding of existing and projected business processes, which can be achieved by good modelling practices, leading to models of business requirements as well as processes and then to well-constructed specifications.

While many companies have invested significantly in SOA and there have been some success stories, there is evidence that these early adopters may not be getting all the benefits that might have been expected. I feel that the need is to return to the basic wisdoms of software engineering, and the approach taken herein emphasizes this, putting a strong emphasis on best practice and business alignment. As new standards for business process modelling such as BPMN and BPEL emerge and evolve, there is also a need for a practical and critical assessment of them. It is particularly apposite to situate this study in the context of organizations moving towards SOA as a strategic direction.

Thus the book covers techniques and notations for requirements modelling, business process modelling and specification: UML, use cases, activity diagrams, Catalysis Conversation Analysis, Mission Grids, BPMN and BPEL. The focus throughout is on SOA. There is also a discussion of applicable standards and technology. A forthcoming companion volume (Graham, 2009) will cover managing SOA projects with agile development practices, governance, skills needed, migration strategy, and so on.

My objectives in writing it are to do the following.

- Provide a basic language-independent introduction to SOA concepts and technology.
- Explicate the business and technology drivers for SOA.

- Offer some new insights into the nature of business processes, especially those that involve human interaction and collaboration.
- Go beyond mere process area modelling and explain how to construct models at every scale.
- Explain the links between BPM and SOA.
- Explain the links between BPM and business rules.
- Provide a comprehensive coverage of modern approaches and notations for requirements modelling and specification.
- Offer and explain some well-tried but possibly unfamiliar practical requirements engineering techniques.
- Present some useful requirements patterns.
- Situate all this in the context of a migration strategy to SOA that is focused on business agility and which will be the subject of the forthcoming book cited above.

Whether I have succeeded, only the reader can judge.

Acknowledgements

There are some people without whom this book, whatever its current defects, would be a great deal weaker. I must thank Derek Andrews, who was the principal developer of Trireme's e-learning course on SOA (www.trireme.com), for many discussions, communications and valuable insights. The first chapter parallels this material quite closely and, I hope, could act as a back-up text for those taking the course. I am similarly deeply indebted to Hubert Matthews for all that he has taught me during long telephone discussions and exchanges of email. Clive Menhinick not only acted the part of a first class technical redactor but made very many vital suggestions as to content and its correctness.

Thanks also to the editorial and production teams at Wiley, who have been very patient and helpful throughout the project.

Ian Graham

CHAPTER

1

Principles of SOA

The physician can bury his mistakes, but the architect can only advise his client to plant vines.

Frank Lloyd Wright (1953)

Computer systems are critical for modern, increasingly global businesses. These organizations continually strive for shorter time to market and to lower the cost of developing and maintaining computer applications to support their operations. However, according to regular reports from the Standish Group between the mid-1990s and the present day, around two thirds of large US projects fail, either through cancellation, overrunning their budgets massively, delivering a product that is never put into production or requiring major rework as soon as delivered. Outright project failures account for 15 % of all projects, a vast improvement over the 31 % failure rate reported in the first survey in 1994 but still a grim fact. On top of this, projects that were over time, over budget or lacking critical features and requirements totalled 51 % of all projects in the 2004 survey. It is not incredible to extrapolate these scandalous figures to other parts of the world. What is harder to believe is that our industry, and the people in it, can remain insouciant in the face of such a shameful situation. Clearly we should be doing something differently.

1.1 Why Projects Fail

The Standish conclusions are further illuminated by the data represented in Figure 1-1, which shows the fate of US defence projects (Connell and

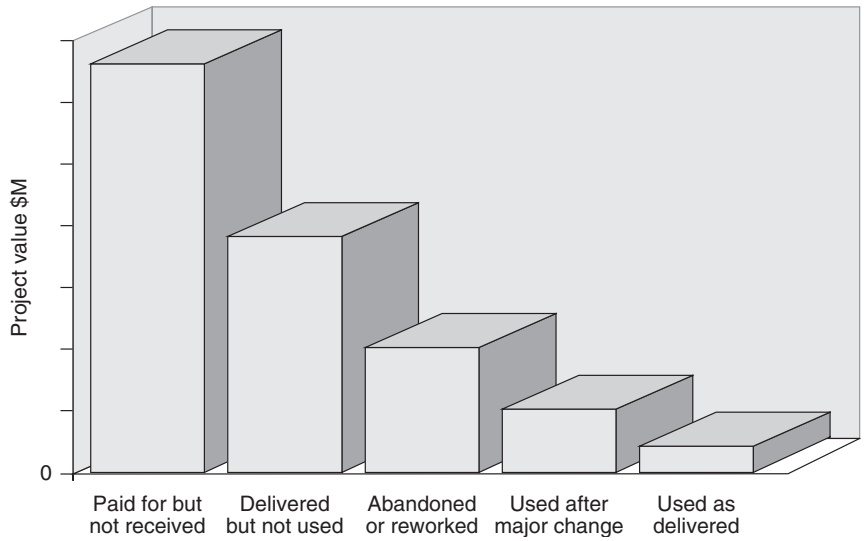

Figure 1-1 The outcome of US defence projects according to US government statistics.

Shafer, 1989). It must be remembered that these systems were mainly mainframe systems written in languages such as PL/1 and COBOL and it may be unfair to make a comparison with systems developed with modern tools. However, the point that something was wrong, even back then, cannot be avoided. Furthermore, the modern evidence seems to suggest that, sadly, not that much has changed.

The Standish surveys also looked into the reasons why people involved in the sample projects thought such projects fail. The reasons given included, *inter alia*:

- lack of user involvement;
- no clearly stated requirements;
- absence of project 'ownership';
- lack of clear vision and objectives.

Why should this be? If the cause really is lack of user involvement (leading to the other three) then we must ask why users are so reluctant. If the system is worth building (and paying for) then, surely, it must be worth spending some time to ensure it does what the user really wants. Is it because they have had bad experiences with IT in the past, perhaps?

Could it be that previous projects involved copious amounts of time spent with that clever C++ programmer (you know, the one with the Ph.D. in Arrogance) poring over huge diagrams that obviously made some sort of sense to him? Could it be that, by the time the system was delivered, the

business had moved on so that changes had to be made and these changes took forever and ramped up the cost 'olympically'. Of course I'm too busy!

There are several reasons why our customers are exasperated with us nice IT folk.

- The typical IT person is more concerned with specification than with modelling requirements. The project manager wants to rein in everything to inside the system boundary and the designers think that a sexy system architecture is cool.
- IT folk do not speak the same language as their users. We speak UML and you, Mr User, must learn it if you want us to be able to communicate successfully.
- The architecture of our systems is driven by fashionable technology and short term project goals. This means that the principles of software engineering best practice are usually ignored in the scramble to cut and test the code. This accounts for the discrediting of the various volleys of silver bullets over the past years: structured design, object-oriented methods, component based development, etc. SOA, in some ways is a repackaging of the same (good) ideas – as we will see in the next chapter.
- Furthermore, the level of abstraction at which we work tends to be far too low.

IT departments are often culturally and technically miles away from the concerns and thought processes of the customers they serve. The problem is, thus, far broader than the need for SOA or any other technological solution; the real problem we have to solve is how to align IT practice with business need and begin to speak a common language. If this can be achieved there is a chance the SOA will not suffer the ignominy of its illustrious predecessors.

1.2 Aligning IT with Business – Speaking a Common Language

To believe that adopting the latest technology fad, be it service oriented architecture, business process modelling or even a business rules management system will, on its own, solve this problem is nothing short of naïve. To align IT with business we must consider all these along with innovative approaches to requirements engineering and system modelling.

The problem of requirements engineering is a modelling problem. We must model the business: its processes, its goals, its systems, its people and structure and even its culture. But we also need to model potential

solutions: in this context, networks of loosely coupled services (applications) that make sense to the business and contribute to its goals. Ideally, the services (the technology, if you must) will map clearly onto the business needs and processes. This implies that we need a language rich enough to describe both the business and its systems and, more importantly, a language that can be understood by all stakeholders.

If one adopts the common misconception that understanding a client's requirements is the same as specifying a system that will meet those requirements, one can then blithely infer that use case analysis is the only requirements modelling technique needed. Jackson (1998) pours scorn on this idea, arguing that use cases are useful for specifying systems but that they cannot describe requirements fully. Use cases connect actors – which represent *users* adopting rôles – to systems. Requirements, on the other hand, may be those of people and organizations that never get anywhere near the system boundary.

In Figure 1-2 we see a depiction of part of Jackson's argument. A requirements document must be written in a language whose designations concern things in the world in which the system is embedded (including of course that system). Specifications need only describe the interfaces of the system and therefore depend on different designations. A specification *S* describes the interface of phenomena shared between the world and the system; use cases may be used to express these. A requirements model *R* is a description over these and other phenomena in the world. *R* depends on both the specification *and* the world. Jackson also states that 'the customer is usually interested in effects that are felt some distance from the machine'.

Ignoring non-user interactions can lead to us missing important reengineering opportunities. I once worked on a rule-based order processing and auto-pricing system, whose aim was to take orders from customers electronically and price them automatically using various, often complex, pricing engines. The problem was that some orders were too complex or too large to admit of automatic handling. These had to be looked at by a salesman who would of course have an interface with the 'system'. So

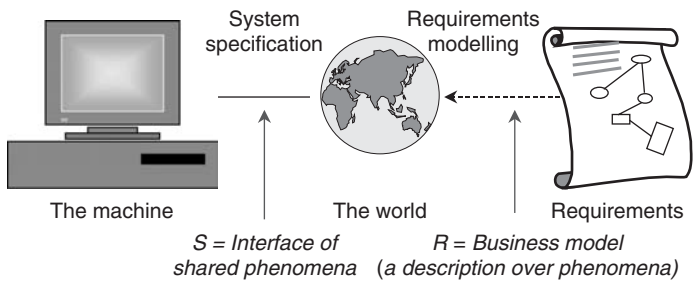

Figure 1-2 Specifications are not requirements models.

far, so good: a rule engine would screen 'illegal' or 'handle manually' orders. The salesman would then apply his various spreadsheets and other routines to such orders. But a further problem existed: some orders were so complicated as to be beyond the skills of the salesman, who did not have expertise in financial mathematics. For these orders, the salesman had to go across the office and talk to a specialist trader. He/she did have the requisite Ph.D. in Financial Engineering. We also modelled this conversation as the 'pseudo-use-case' shown as '`Help!`' in Figure 1-3 and, as a result, when our domain expert looked at the simulation we had built, he/she realized immediately that if we gave the trader a screen (for orders re-routed) we could radically improve the workflow, and thereby customer service. Even this relatively minor excursion away from the system boundary thus had a big cash import because the orders for complex products were precisely the most profitable orders – by a long way. In many, more complex cases, the importance of going beyond the boundary will be greater still. Jackson gives an example of patient monitoring wherein sensors are attached to a patient's vital signs, and alarms (i.e. variances from tolerance) are forwarded to a nurse's workstation. The problem is that, should an alarm be triggered, the nurse is not normally the actor who will save the patient's life. She must run down the corridor to fetch a doctor. Thus, the critical use case is not at the system boundary and would be ignored in the conventional approach. Put starkly, concentrate on the use cases at the system boundary and people may die.

My interpretation of Jackson's argument is that we need a specific technique for modelling business processes distinct from, but compatible with, use case models of specifications. The alternative is to fall back on a veritable 'Russian doll' of nested models described in terms of 'business

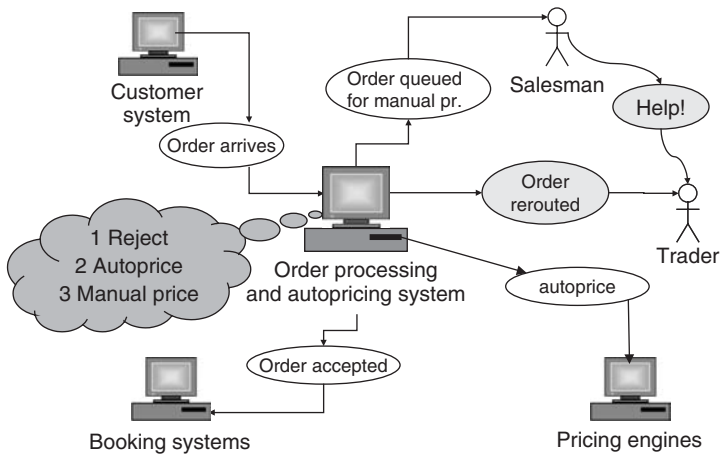

Figure 1-3 A process for order processing.

use cases' (Jacobson *et al*., 1995): an approach that is not only clumsy but fails to address the above arguments. Thus, we need to know the answers to the following two questions before we can proceed.

- What is a model?
- What is a business process?

We defer dealing with the second question until a later chapter. Later in this book we will explore various notations and approaches to business process modelling. Let us first look at the nature of models.

1.2.1 Models

Modelling is central to software engineering practice and especially within the context of service oriented architecture. A **model** is a representation of some thing or system of things with all or some of the following properties.

- It is always *different* from the thing or system being modelled (the *original*) in scale, implementation or behaviour.
- It has the shape or appearance of the original (an iconic model).
- It can be manipulated or exercised in such a way that its behaviour or properties can be used to predict the behaviour or properties of the original (a simulation model).
- There is always some correspondence between the model and the original.

Examples of models abound throughout daily life: mock-ups of aeroplanes in wind tunnels; architectural scale models; models of network traffic using compressed air in tubes or electrical circuits; scaled models of silting up in river estuaries; software models of gas combustion in car engines. Of course, *all* software is a model of something, just as all mathematical equations are (analytic) models.

Jackson (1995) relates models to descriptions by saying that modelling a domain involves making designations of the primitives of the domain and then using these to build a description of the properties, relationships and behaviour that are true of that domain. For example, if we take the domain of sending birthday cards to one's friends, we might make designations:

```
p is a friend;
d is a date (day and month);
B(p,d) says that p was born on d.
```

Then we can make descriptions like: for every p, there is exactly one B. Jackson suggests that modelling is all about ensuring that the descriptions

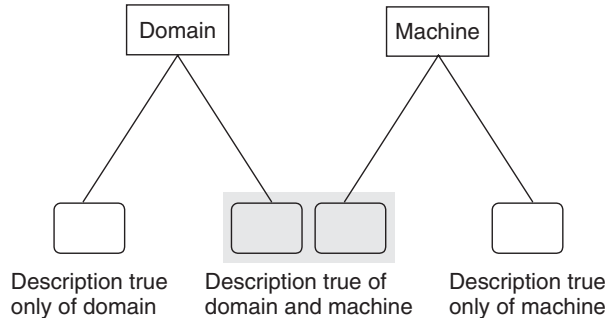

Figure 1-4 M is for model (after Jackson, 1995).

apply equally well to the model and to the original domain. In the context of computer models this might mean that the instances of a class or the records of a database are made to correspond uniquely to domain instances of our friends. Most usefully, Jackson presents this concept as the M configuration shown in Figure 1-4.

The Domain and the Machine are different; in the domain friends do not reside in disk sectors. There are many things in the domain that are not in our model, such as our friends' legs or pimples. There are also things that go on in computers that we are not concerned with in a model, such as load balancing. The model comprises the features shared between the domain and the machine.

This understanding of what a model is can be applied to the problem of service modelling. We must understand clearly that a service is both a model of the domain and a potentially implementable machine model. But we must begin with a model of the domain to understand and validate the requirements.

This understanding leads also to another common difficulty with computer systems: they can get out of synch with the world. The most topical example of this is probably identity theft; the computer model thinks you live somewhere else! It is therefore a key requirement of most systems that there should be a mechanism for re-synching the models should they become out of kilter.

Good models are abstract but 'real'; they ignore unnecessary detail but correspond exactly to the true state of the business. Good models are re-synchable. Good models are understandable to non-technical people.

Applying this understanding to modelling services, a service is derived from a model of the business domain and of the potential implementation domain. Defining the services precisely is the only way we can be sure that they meet current needs. Modelling unifies and clarifies those services and allows us to know what they are. Precise definition also provides a contract for the software developers.

Note that services may have relationships between them and often do: this service can only be accessed after that one; these services can be done in parallel; this service enables or disables that one; etc.

These relationships are about business rules. If we change the relationship, we will change the way we do business or re-engineer the business. Such an approach helps us to integrate different parts of the business and be more agile. It also supports greater reuse because we can know exactly what the service is. Additionally, it helps identify services required from other businesses that, hitherto, we might have been forced to create from scratch, thus duplicating them.

In later chapters we will return to the question of how to construct a sufficiently rich language for business modelling that is equally understandable to the user and to IT staff. We will focus on techniques for modelling requirements, system and business processes specifications on the way to delivering an SOA that can support them properly and allow for flexible and low-cost evolution. But first we must understand the principles and basic concepts of service oriented architecture.

1.3 What is Service Oriented Architecture?

Service oriented architecture (SOA) is an architectural concept in software design that emphasizes the use of combined loosely coupled services to support business requirements directly. In SOA, resources are made available to service consumers in the network as independent artefacts that are accessed in a standardized way. This adherence to standardization is definitional. SOA is precisely about raising the level of abstraction so that requirements and business processes can be discussed in a language understood by business people as well as IT folk.

The main idea behind SOA is the desire to build applications that support a business process by combining a number of smaller 'business services' into a complete business process or workflow. Each of these services is a stand-alone piece of software providing business functionality that is loosely coupled to the other services (other pieces of software) which make up the application. Examples of a business service could be checking details about a customer, validating a customer payment, sending an invoice to a customer, synchronizing or transferring data between systems, or converting a document from one format to another. Many of these services will be particular to a business; however, some will also be standard services that could either be purchased as software or will be readily available on the internet in the form of web services. New services can also be created from existing applications or by writing new ones using your preferred development framework.

Many definitions of SOA identify the use of web services (using SOAP and WSDL) in its implementation; however it is possible to implement SOA using any service-based technology. Though built on similar principles, SOA is *not* the same as web services. SOA is independent of any specific technologies.

A software architecture is a representation of a software system – how all the pieces fit together. It describes the most effective way to design the system within a set of constraints or a defined infrastructure. An architectural style is a family of architectures sharing common themes and a recognizable common vision, in the same way that we can recognize the shared vision of Gothic or Georgian architecture. Service oriented architecture is an architectural style whose goal is to achieve loose coupling among interacting software agents. A software agent is an application, a piece of software, a component, a program, etc; also known in older terminology as a module – a piece of software that does work. A **service** is a unit of work carried out by a **service provider** to achieve some desired result for the **service consumer**. Typically, a unit of work would be some sort of (business) transaction. The service consumer has a goal in mind, and the task of the service provider is to achieve that goal on behalf of the consumer or help the consumer to do so. Both provider and consumer are rôles played by (possibly software) agents.

Although a service is just an interface, it will thus be implemented by a software component, or collection of software components, that implements a reusable business function, such managing customers or managing customer accounts. Services are defined by their interfaces, which describe what the services can do. How the service works is hidden inside the component or components that provide the service.

Service-oriented architecture is an approach for designing and building applications by constructing them from loosely coupled services (existing services where possible) and discovering and writing new services as necessary.

The SOA approach encourages loose coupling between the services that make up an application; contrast this to traditional monolithic architectures, which are characterized by tight interdependence between the parts of an application.

Figure 1-5 illustrates how a user might interact with a product ordering or quotation service. The user asks a question and gets a straightforward and useful answer. To assemble this response the system actually relies on four lower level services, including one which could well be rule-based, since tax regulations vary quite often.

The most important properties of a service are as follows.

- Every service has a **contract**: a description of what the service will do for the user and what it requires of the environment and indeed the

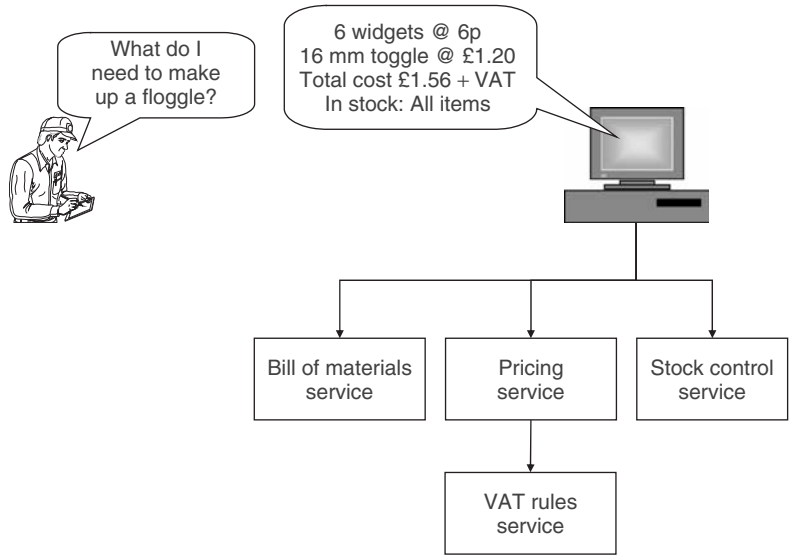

Figure 1-5 Composing loosely coupled services.

user. This is what enables the loose coupling between services and is the basis for 'discovering' [services over a distributed environment]. Clearly defined contracts are essential if we are to achieve composition and reuse of services.

- Services can be **discovered**. They are designed to have a description in a directory, so that they can be found and accessed via a discovery mechanism. This too helps make services more readily reusable. Discovery is a property that can be given to any piece of software – but it doesn't necessarily mean anyone will find that software actually useful.
- Services are **abstract** – the only part of a service that is visible to the outside world is the service description (the contract). This contributes greatly to service reusability or sharing.
- Services are **autonomous**: they have control over the logic they encapsulate; they decide how any arriving messages should be processed or forwarded. This enables service composition and makes reuse easier to achieve.
- Services can be **composed** – they can be combined with other services to satisfy a set of business requirements, to solve new problems.
- Services should be **loosely coupled** to other services. This enables composition and encourages autonomy.
- Services are **stateless with respect to complete transactions**. Services minimize the storage of information specific to an activity (a use of

the service). This too helps with composability. A stateless service is an ideal that should always be strived for, but we can weaken this requirement by passing the state either directly or indirectly with any message exchange.

- Services should be **reusable**. Systems are best divided into services with the conscious intention of promoting reuse.

One needs to focus on precisely these eight properties of a service when developing a new service or identifying and wrapping existing applications as services. Looking closely at these, we can see that reuse depends on all the other properties.

- Abstraction helps with packaging for reuse.
- An autonomous and stateless service is more likely to be reusable.
- Loose coupling means minimizing dependencies on other software and this encourages reuse.
- A service that is discovered can be used, and used again; it can be reused.

Also, adversely:

- Reuse takes place when a service is composed with other services.
- A service that is autonomous and stateless is easier to compose with other services.
- It is precisely loose coupling that allows composition.

Thus both reuse and composition result from the other six properties.

There is an even more important property that we have not listed here. As shown in Figure 1-6, a service must supply a service to the business. This means that the service does something that achieves a business goal. It will be about a sale, identifying a customer, reserving a seat on an aeroplane or some such. The service will not be about supporting functions in a user interface.

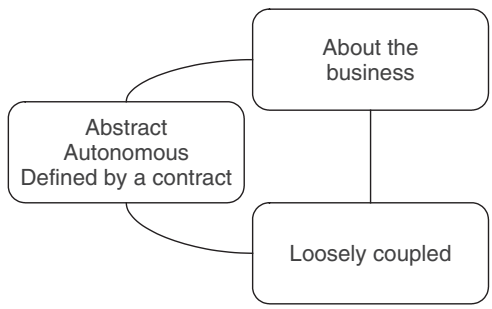

Figure 1-6 The key properties of a service.

The interface of a service, as presented to the user, displays a set of operations that make up that service. These operations are about supporting the user of the service. But who *is* the user of the service? The usual answer is an actor (using UML use case terminology). An actor is a user adopting a rôle or another system that interacts with the service. However, it is necessary to think more carefully about this. Services are about supporting the business, not supporting a computer user or another system. The service should support something or somebody *in the business*, in the real world: the **real user** of the service. The real user, in this sense may be someone who never comes anywhere near a computer or browser; someone who is far away from the system boundary.

The operations of a service should be abstract and at a high level of abstraction and be about the business. Some examples will illustrate this point.

When reserving a hotel room, say, we need to create a record with all the necessary details obtained from the real user, the person making the reservation; rather than

```
get guest details (name, address, contact tel)
get reservation details (room type, arrival date, days)
get reserver details(name, address, contact tel)
```

These are probably operations supporting a user interface or procedure calls in some programming language. Instead of this, a service invocation (message) should contain all the information that is needed for a coherent transaction to be possible. For example:

```
message<guest name, guest address, guest contact
tel, room type, arrival date, days, reserver name,
reserver address, reserver contact tel>
```

SOA emphasizes building systems for the user, not systems for the IT department: systems for the benefits of employees, customers, suppliers, partners; systems for the 'real' user!

In the past, clerks used 'terminals' to access 'batch systems'; now customers use the web . . . or do they?

I was once involved in the design of a web radio station. Our approach soon teased out the fact that the most important business objective behind the initiative was neither to do with entertainment, any Reithian concern with edification nor with selling advertising (they would then have only lost the same revenue from the airtime ads). No, the main purpose was to sell CDs. The target audience was teenagers, so the strategy included providing lots of contents concerning current pop stars and bands. Here is one of the scenarios we explored during one requirements workshop.

Denise Green is visiting her friend Tracey Black. They are eating chocolate and discussing music and decide to log on to the station to check out some

groovy sounds and get all the latest poop on their favourite boy bands. To their delight, BoyzInCustard have a new album out today. They *must* have it. So far so good. But there is a problem; neither girl is old enough to have a credit card. Never mind, good old Mum's got one. Mrs Black wouldn't go near a computer if her life depended on it. However, she is precisely the real user so far as the financial part of the radio station's business process is concerned.

Figure 1-7 provides another way to visualize this point.

Loose coupling is about having a simple interface with very low dependency – an interface that mandates collaboration with 50 other services in order for it to do its job and for them to do their jobs would be impossible to extract for reuse. To say that services can be loosely coupled implies that the service interface must be independent of the implementation. There can be unintended consequences of coupling (both good and bad ones). Ideally, services are 'stateless'; which means that, when a message arrives at a service interface, the service processes the message and returns an answer or result to the consumer; it remembers nothing about the message afterwards; it does not even need (apart from the reply address) to know anything about the consumer.

A service can receive a message, process it, return a reply and then forget all about it. At any rate, this is all we can see from the outside. Looking inside the service, however, we might see three types of communication ports or 'endpoints'.

- Ports for messages entering the service; the **entry** ports.
- A departure port for sending the reply message; the **exit** port.
- A **rejected message** port for invalid messages.

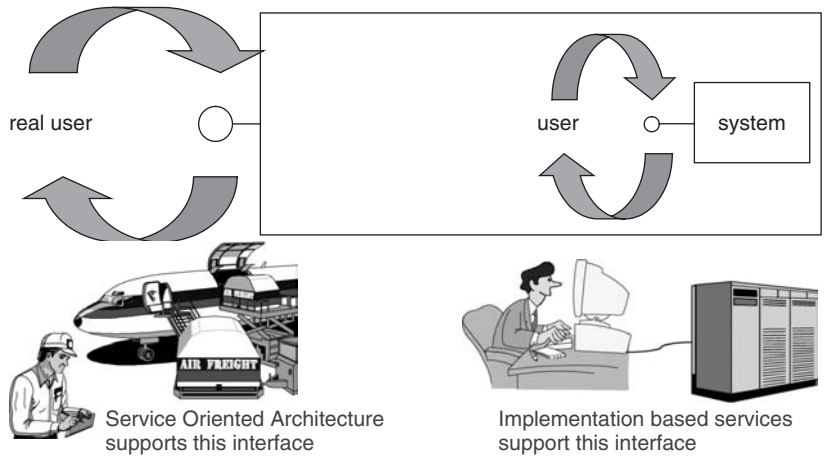

Figure 1-7 Services for customers not clerks.

There could be more ports such as those for queuing application faults or tracking messages, but the above three are the basic ones.

A service will read a message from the entry port, process it and write the reply to the exit port. If the message is invalid (it is not recognized by the service), it is written to the rejected message port. Developing a service in this style is reasonably straightforward. The service knows nothing about the outside environment, it is entirely self-contained and all it needs to do is process a message it recognizes, reject a message it doesn't, send any reply as necessary, and then forget about the message.

Note that this type of service will probably need a complex environment it can exist in. The trick is to make this complex environment easy to configure and use. As we shall see later, enterprise service bus products offer one such configurable environment.

In summary, the perfect service is a black box, loosely coupled to all users. Its overall action is input–process–output, with no memory of the input after producing the output.

It is vitally important that a service can be extended. Businesses change rapidly and so it is important that software changes to reflect this. This will involve changes to service providers, service consumers and the messages. If these changes cannot be made, then everyone is locked into the current version of a service, it will not be possible to extend it to reflect the new business opportunities. What changes can be made to a service without invalidating it for current users?

We can change the implementation providing the interface is still satisfied, we can change the interface by adding function of accepting more types of input and we can add fields to a message. Changes other than these are likely to invalidate or corrupt the service for existing users.

We obtain optimal loose coupling when a service is a perfect black box, in the sense that any user cannot see inside it; and, when once inside the service, you cannot see out; so that the service has no idea of who is using it and what other services are around. Making sure that services really are black boxes really does reduce coupling.

Each SOA service should have a quality of service (QoS) associated with it. Typical QoS elements are security requirements, such as authentication and authorization, reliable messaging, and policies regarding who can invoke services. QoS statements may include SLAs, but in principle they can include more than mere service level agreements. I will give an example of this.

Let us suppose that our quotation service needs to vary the price of widgets dynamically as the open market price of copper fluctuates. This price must be obtained from an on-line information provider such as Reuters or Honest John's Prices Inc. as shown in Figure 1-8. Honest John provides a low cost option but, let's face it, probably isn't anywhere near a reliable as a reputable or well-established firm such as Reuters or

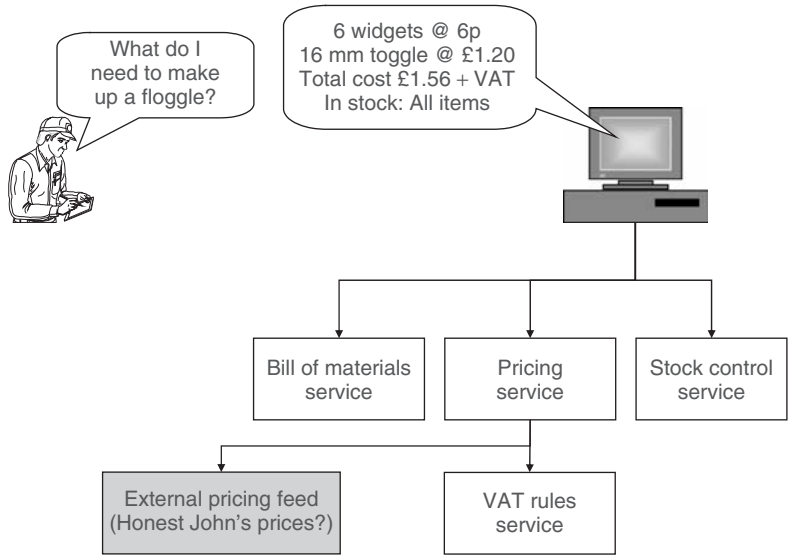

Figure 1-8 External services must be trustworthy.

Thompson (Datastream). We must not know or care about how such a service is implemented but we might well want to specify a level of 'trust' or some surrogate for that quality.

There are other, more mundane, meanings for QoS. The term is most often used to refer to qualities like security or the reliability of message delivery. These are important, of course, but a good service needs to be trusted in every sense of the word.

Application developers or system integrators can build applications by composing one or more services without knowing the services' underlying implementations. For example, a service can be implemented either in .NET or J2EE, and the application consuming the service can be on a different platform or language.

SOA, if done properly, should lead to interfaces that are about the business and not about supporting the user interface; there is end-to-end involvement with both customers and suppliers. Beware especially of just wrapping existing interfaces; this is usually far too low level. Make sure, when modelling, that you understand the business, not just the computer systems. Provide services for *people* to do tasks that deliver them value, whether these people are employees, customers, suppliers, regulators or any other kind of stakeholder. Furthermore, they should be able to understand the system in their own terms rather having to learn the software developers' argot.

SOA should provide services that help people carry out tasks that deliver them value; systems for the user not for the IT department; systems for