Designing and implementing SOA-based solutions

Service-oriented architecture using Web services, JBI, WS-BPEL and URDAD

COLLABORATORS

	TITLE : Designing and implementing SOA-based solutions		*REFERENCE* :
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Dawid Loubser and Fritz Solms	February 13, 2009	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

Preface: About Solms TCD

Copyright

Overview

The training pillar of Solms TCD focuses on vendor-neutral training which provides both, short- and long-term value to the attendees. We provide training for architects, designers, developers, business analysts and project managers.

Vendor-neutral, concepts-based training with short- and long-term value

None of our courses are specific to vendor solutions, and a lot of emphasis is placed on entrenching a deeper understanding of the underlying concepts. In this context, we only promote and encourage public (open) standards.

Nevertheless, candidates obtain a lot of hands-on and practical experience in these open-standards based technologies. This enables them to immediately become productive and proficient in these technologies, and entrenches the conceptual understanding of these technologies.

When a ‘product’ is required for practicals, we tend towards selecting open-source solutions above vendor products. Typically, open source solutions adhere more closely to public standards, and the workflows are less often hidden behind convenient wizards, exposing the steps more clearly. This typically helps to gain a deeper understanding of the appropriate technologies.

Training methods

We provide instructor based training at our own training centre as well as on-site training anywhere in the world. In addition to this we provide further guidance in the form of mentoring and consulting services.

Instructor-Led Training

Our instructors have extensive theoretical knowledge and practical experience in various sciences and technologies.

After familiarising ourselves with the individual skill and needs of candidates, we adapt our training to be most effective in the candidate’s context - deviating from the set course notes if necessary.

Students spend half their time in lectures and half their time in hands-on, instructor-assisted practicals. Class sizes are typically limited to 12 to enable instructors to monitor the progress of each candidate effectively.

About the Author(s)

Fritz Solms

Fritz Solms is the MD and one of the founding members of the company.

Besides the management role, he is particularly focused on architecture, design, software development processes and requirements management.

Fritz Solms has a PhD and BSc degrees in Theoretical Physics from the University of Pretoria and a Masters degree in Physics from UNISA. After completing a short post-doc, he took up a senior lectureship in Applied Mathematics at the Rand Afrikaans University. There he founded, together with Prof W.-H. Steeb, the *International School for Scientific Computing* - developing a large number of courses focused on the immediate needs of industry. These include *C++*, *Java*, *Object-Oriented Analysis and Design*, *CORBA*, *Neural Networks*, *Fuzzy Logic* and *Information Theory and Maximum Entropy Inference*. The ISSC was the first institution in South Africa offering courses in *Java* and the *Unified Modeling Language*. During this period he was also responsible for presenting the OO Training for the Education Division of IBM South Africa.

In 1998 he joined the Quantitative Applications Division of the *Standard Corporate and Merchant Bank* (SCMB). Here he was the key person developing the architecture and infrastructure for the QAD library and applications. These were based on Java and CORBA technologies, with a robust object-oriented analysis and design backbone.

In 2000 he and Ellen Solms founded Solms TCD.

E-Mail:	`fritz@solms.co.za`
Tel:	011 646 6459

Dawid Loubser

Dawid Loubser is a director of Solms TCD, and a specialist in Java, XML, and various graphics- and media related technologies across a range of application domains. He has a long history in the field of architecture and design, and a strong interest in Semantic Web technologies and user interface design.

After working at the JSE (Johannesburg Stock Exchange) as architect and lead developer for web-based systems, he joined Solms TCD to persue a broad and varied journey of training, consulting and development. He presents courses on various SOA, Java and XML-based topics, and is actively involved in research around the Java language, XML Schema, and is co-creator of the URDAD analysis and design methodology.

E-Mail:	dawidl@solms.co.za
Telephone:	+27 (11) 646-6459

Solms TCD Guarantee

We hope that the course will comply with your expectations, and hopefully exceed it! Should you for some reason you feel that you are not satisfied with

- the course content,
- the teaching methods,
- the course presenter or
- any auxillary services supplied

Please feel free to discuss any complaints you may have with us. We will do our best to address your complaints. Should you feel that your complaints are not satisfactorarily addressed within our organization, then you can raise your complaints with:

- Professor W.-H. Steeb from the *University of Johannesburg*, Tel: `+27 (11) 486-4270`, E-Mail: `whs@rau.ac.za`
- Dr A. Gerber from the Computer Science Department of the *University of South Africa*, E-Mail: `gerberaj@unisa.ac.za`

At the time of writing we are in the process of obtaining *ISETT SETA* accreditation. Complaints can be raised directly to that institution.

Chapter 1

Introduction

1.1 The Context of this course

The current technology landscape finds itself in a rather precarious position: There has hardly ever been a time of greater technological diversity within our systems. While the support for public standards (on all fronts, such as network and messaging protocols, component models, handling of security and transactions, etc) is very strong, different systems exist to satisfy different stakeholder requirements; and thus continue to be built in isolation using different implementation technologies.

Furthermore, almost all attempts to standardise the functional aspects of systems (and by this, we mean the structure of stored / exchanged data, and the nature of the services provided by the system) have proven to be futile, and rightly so: Different stakeholders within (our outside of) the organisation require different services.

In this complex and non-standardised environment, we need to be able to integrate and orchestrate different systems. The aim of this course, is to illustrate how SOA (Services-Oriented Architecture) provides a compelling solution to these problems. When it comes to SOA technologies and middleware, we feel that it is especially important to stick to public standards, to avoid vendor lock-in in terms of tools and infrastructure (one of the core risks of SOA). Unfortunately, this means that we often forgo the luxury of all-in-one (often graphical) toolkits, having to often delve deeper to understand what really happens. The positive side of this situation is that we believe the material covered in this course will be valuable, both theoretically and practically, regardless of the ultimate SOA technology solution chosen by the candidate.

We promote a Model-Driven systems development process (as per the URDAD methodology) where the business analyst models the solution to a problem domain completely (across all levels of granularity) in a technology-neutral manner, which is subsequently *mapped* to a particular implementation domain. Parts of the solution may be mapped to systems, and parts may be mapped to humans.

1.2 Target audience

This course is targeted at developers who are tasked with the implementation of an SOA-based solution. In addition to covering the most important technologies to realise this goal, we also assist the developer in mapping a URDAD/UML-based design to common SOA technologies.

Though many of the abstract SOA technologies and concepts are not associated with any particular programming language or software framework, this course promotes Java-based implementations of both the SOA frameworks as a whole (via standards such as JBI) as well as individual business-logic components (via standards such as EJB and JAX-WS). Java is thus favoured for the places where we require an imperative programming language.

Part I

SOA Concepts

Chapter 2

The Services Oriented Enterprise (SOE)

2.1 Introduction

Increased global competitativeness and regulatory compliance has resulted in an environment which requires companies to be

- *more flexible*, such that they can more rapidly exploit market opportunities and address changing clients,
- *leaner*, in that they focus on their core business and can efficiently make use of external service providers to address those responsibilities which do not fall within its core business focus,
- *more transparent*, in that the business processes can be readily monitored and inspected.

Historically many large organizations have been modelled around the hierarchical pattern. Even though the movement has been towards "flattening organizations", this mainly resulted in reducing the number of layers in the hierarchy without moving away from the hierarchical structure as such. Hierarchical organization often focus more on structure and accountability than on core focus delivery and business processes.

A Services Oriented Enterprise (SOE), on the other hand, is more process and less structure focused. In a SOE, all activities in the organization are are focused on service delivery and value generation. In a SOE, every organizational component is viewed as a provider and potentially also consumer of services. It sees every element of the organization from a service provider and consumer perspective. This shifts the focus of all aspects of the organization towards stake holder value generation and results in an organizational infrastructure which is more flexible, leaner and more transparent.

The services sector has for some time been the dominant sector of most advanced economies. But, just because a company is a services company, it does not mean it is a Services Oriented Enterprise (SOE).

2.1.1 There is only a services sector

All organizations are, in principle, operating within the services sector. Even manufacturing companies ultimately provide product manufacturing services even though the revenue is obtained from selling products. This is, however, no different to traditional services companies packaging a collection of their services within a "product" and selling them as a product.

For example, telecommunication providers often package their telecommunication services within products like data bundles.

2.2 What is a Services-Oriented Enterprise (SOE)?

A Services Oriented Enterprise (SOE) is an organization which is globally integrated and whose core focus is on service delivery and effectively managing the processes around service delivery.

Just because an organization operates within the services sector does not mean that it is a services-oriented enterprise. A SOE should satisfy a number of requirements:

- Any activity done within any organizational unit at any level of granularity is part of realizing a service. Higher level, more coarse grained services are assembled from lower level, more fine grained services.
- Business processes are assembled from services available to the organization, i.e. services provided by internal business units and systems as well as services provided by external service providers.
- There are services contract for the services sourced from internal business units as well as for those sourced from external service providers.
- For any required service, the organization can change the preferred service provider with another service provider which realizes the same services contract. Changing from one service provider to another does not require any changes to the business processes for which these services are required.
- The organization has mechanisms in place for effective service governance.
- A SOE manages an infrastructure through which service requests are routed to the service providers chosen to realize the required services.
- Business processes are defined in a technology-neutral manner, and are managed effectively by business (and not by technology).
- In a SOE, structure is driven from processes and continuously optimized in the context of business process optimization. The sole purpose of any structural elements is that of supporting business processes which generate value for the stake holders of the organization.
- A SOE typically aims for real-time service discovery and dynamic service provider selection.

2.2.1 SOE and SOA

A Services-Oriented Enterprise (SOE) need not necessarily be built around a Services Oriented Systems Architecture (SOA). However, the process of aligning the *systems architecture* with the *organizational architecture* will typically lead to a SOE choosing an SOA as core element of the systems architecture.

An SOA will enable the SOE to automate a wide range of responsibilities and tasks including

- business process execution,
- static and dynamic service provider discovery and selection,
- integration with service providers, service request routing and delivery, and
- service governance.

2.3 Typical structure of a Services Oriented Enterprise (SOE)

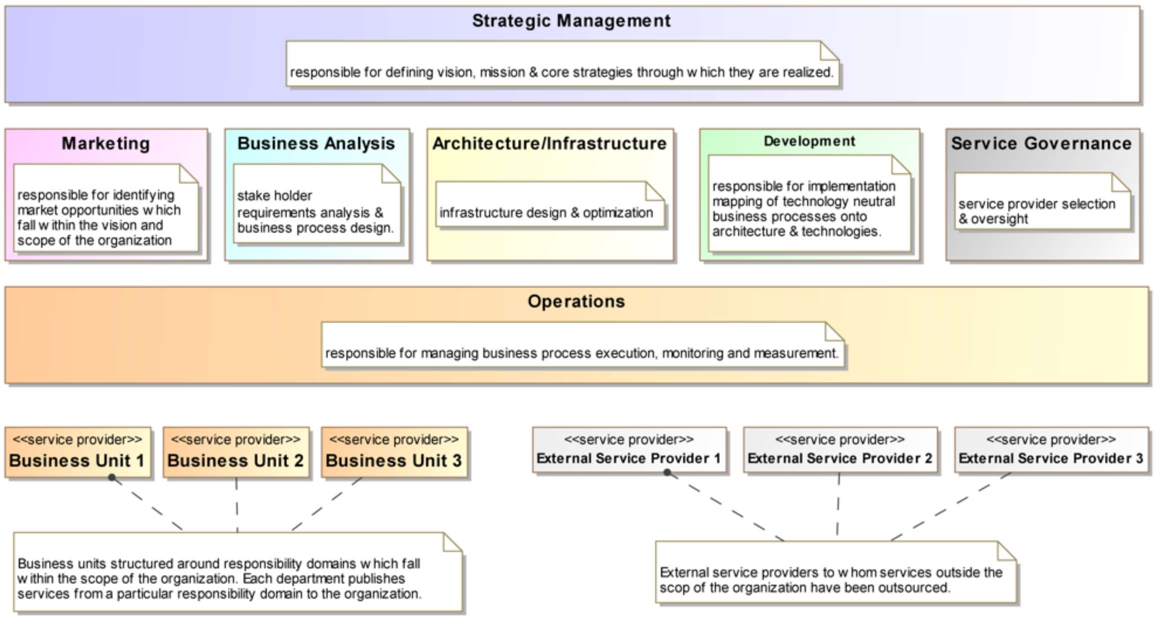

Strategic Management
responsible for defining vision, mission & core strategies through which they are realized.
Marketing
responsible for identifying market opportunities which fall within the vision and scope of the organization
Business Analysis
stake holder requirements analysis & business process design.
Architecture/Infrastructure
infrastructure design & optimization
Development
responsible for implementation mapping of technology neutral business processes onto architecture & technologies.
Service Governance
service provider selection & oversight
Operations
responsible for managing business process execution, monitoring and measurement.
<<service provider>> Business Unit 1
<<service provider>> Business Unit 2
<<service provider>> Business Unit 3
<<service provider>> External Service Provider 1
<<service provider>> External Service Provider 2
<<service provider>> External Service Provider 3
Business units structured around responsibility domains which fall within the scope of the organization. Each department publishes services from a particular responsibility domain to the organization.
External service providers to whom services outside the scop of the organization have been outsourced.

Figure 2.1: Typical structure of a Services Oriented Enterprise

2.4 Governance in an SOE

One of the critical aspects of a services-oriented enterprise is that of governance. Governance is concerned with oversight, monitoring, reporting and enforcing of the organization's policies and processes. Governance is critical for

- ensuring that the organization's activities are aligned with its vision and mission,
- quality assurance across the organization's processes,
- managing the risk the organization is exposed to in the context of both, internal service delivery and external service provider usage,
- optimizing the organization's processes and resource usage in order to maximise stake-holder returns.

2.4.1 Alignment of provided services with organization's vision, mission and business strategies

This includes

- continuously monitoring that the organization's services portfolio includes only services which are aligned with the organization's evolving vision and mission, and
- ensuring that the processes for prioritizing the iterative development of new services in line with the organization's business strategies and optimization of value generation are in place and are adhered to.

2.4.2 Maintaining and monitoring the services catalog

Governance requires the maintenance of a services catalog listing both, the services provided by the organization as well as the services consumed by the organization. Governance will also monitor the services catalog

2.4.3 Architecture/infrastructure management

Governance is responsible for monitoring that the SOE architecture provides a suitable infrastructure for hosting the organization's services and executing the organization's business processes. It needs to ensure that the processes for defining, implementing and evolving the organizational architecture are put in place and adghered to.

2.4.4 Service development management

This ensures that the processes of developing new services are effectively managed and includes management and oversight of

- the service requirements analysis,
- the business process design,
- assessing and the infrastructure requirements for the service are met,
- service construction including software development and personal training processes,
- quality assurance around service development, and
- the processes around service publication and advertizing.

2.4.5 Contract delivery oversight

Governance is responsible for monitoring contract delivery across internal and external service providers and for enforcing processes which address issues around contract delivery.

2.4.6 Oversight, monitoring and reporting of client and stake holder value generation

Governance includes the monitoring of client and stake holder value generation (i.e. customer satisfaction) and the definition and enforcing of processes which optimize these.

2.4.7 Defining and enforcing criteria for service provider selection

Governance must ensure that the criteria for service provider selection are defined, documented, communicated and enforced across the organization.

2.4.8 Enforcing security policies

Governance needs to ensure that the security policies are defined, implemented and adhered to. This includes an infrastructure for authentication as well as the infrastructure and processes around access control which enforce that the entities only obtain access to those services which they should have access to.

2.4.9 Change management

Governance must make certain that the change management processes are put in place and enforced.

2.5 Business process design in the context of a Services Oriened Enterprise

It is paramount that any business process design approach for a Services Oriented Enterprise needs to

- be technology neutral,
- specify the linkage between functional requirements and services through which they are realized,
- churn out services contracts for the responsibility domains from which services are required including
  - the inputs and outputs for each service,
  - the pre- and post-conditions for the service as well as potentially some invariance constraints,
  - the required qualities of service.
- Specify how the organization assembles the business process across the services it sources from internal and external service providers,
- specify the requirements for any exchanged value or data objects,
- specify how service request inputs are assembled from the information available to the organization, and
- for services to be realized within the organization, the recursive assembly of higher level services from lower level services.

Technology neutral, services oriented analysis and design methodologies like URDAD (the Use-Case, Responsibility Driven Analysis And Design methodology) generate a formally-specified, technology-neutral design with the following attributes:

- Responsibilities are assigned to services contracts, and not to concrete service providers or systems.
- Services are assembled recursively from lower level services.
- Full services contracts specifying the functional and quality requirements for all services across levels of granularity are generated.
- Business process logic is removed from the actual service providers and localized in a controller. The controller decouples the service providers from one another and the service providers are not aware of the business process for which the service is required nor of any other service providers participating in the business process.
- Data structures for the exchanged value objects are specified in a technology neutral way and can be mapped onto various implementation technologies including paper, web or GUI based forms, XML or Java objects, ERD diagrams specifying the implementation mapping for relational databases, ...

2.6 Business Intelligence

Business intelligence is used for business decision support around strategic management, marketing and optimization of the organization's operations.

An SOE provides the infrastructure and in particular the underlying SOA systems infrastructure makes it easier to collect metadata around service and resource usage and can make this information as well as reports based on this information available in real time. In particular, they facilitate the real-time calculation of Key Performance Indicators (KPIs) for the organization.

2.6.1 Data warehousing

Traditionally business intelligence has been created around data warehousing. This makes a range of raw and processed information retrieved from Business Processs Management (BPM) Systems, Enterprise Resource Planning (ERP) systems available for analysis and reporting purposes.

The information is typically processed by managers using Online Analytical Processing (OLAP) tools.

2.6.2 Knowledge repositories

As society makes the transition from an information age to a knowledge age, the systems themselves will not only feed data into a data warehousing system, but will contextualize any information with meaningful (semantic) metadata, obtained through the understanding of the business processes themselves, i.e. the systems themselves will actively contribute to the knowledge growth within the organization.

This will not only significantly increase the value which can be obtained from the business intelligence support, but will make it simpler to automate core business decisions themselves. A semantic knowledge repository provides not only significantly richer and contextualized data, it also provides an environment facilitating automated semantic reasoning and real-time systems-based innovation.

2.7 Business benefits and strengths of a Services Oriented Enterprise

Evolving an organization into a Services Oriented Enterprise aims to provide a number of benefits to the organization which ultimately contribute to the organization being able to realize increased stakeholder value. The core benefits envisaged from evolving toward a services-oriented enterprise include the following:

- **Culture of service delivery** A SOE grows a culture of service where each business unit is focused on service delivery and value generation. Business units are service providers, service consumers and at times observers.
- **Contracts-based approach** In a SOE, the focus is on clear formulation of services contracts which specify the functional requirements for services as well as the quality requirements. The services contracts
  - formalizes client requirements,
  - localizes accountabilities,
  - facilitates testability and service provider performance monitoring,
  - facilitates service provider pluggability resulting in lower cost and reduced business risk, and
  - improves reliability.

  The services contracts specify both, the functional and the non-functional (i.e. quality) requirements. The functional requirements define what needs to be provided and the quality requirements specify the required services qualities including requirements around
  - reliability,
  - scalability,
  - performance,
  - service accessibility, integrability and delivery channels,
  - cost constraints, ...
- **Improved reuse** As services the organization's business processes are assembled across self-contained, contract-based services, a SOE is in a position to achieve a high level of reuse, of services available within and outside the organization. This improves responsibility localization and organizational focus, making the organization more lean and mission-focused.
- **Effective service governance** Effective service governance through contracts management, managing preferred service providers for services contracts, as well as through improved monitorability and auditability of service delivery.
- **Competitative business units** A SOE provides an infrastructure within which internal business units need to continuously compete against external service providers, ensuring they remain cost effective and competitative.
- **Globally integrated organization** A services oriented enterprise views itself as fully integrated within the global market, continuously sourcing services from the market and updating the servcices portfolio it provides to the market. In this context, SOEs represent virtual, extended enterprises which are globally connected. They operate within an infrastructure which caters for the assembly assembly business processes across enterprises, i.e.
  - a business process can be easily assembled from services which are available to the organization including services which are realized within the organization itself and services which are sourced globally, and

  - the services offered by the organization can be easily incorporated within business processes of other organizations.

  In either case the services reused could be automated within systems or realized manually.

- **Improved flexibility and time-to-market** Flexibitlity and time to market are driven by the ease with which new business processes can be assembled from services available to the organization, the high level of reuse and the ease with which business processes can be modified.
- **Not structure focused** In an SOE, the focus is on service delivery. Any structure elements explicitly only serves to support the business processes realizing the services. This results in an organizational structure of minimal complexity and overheads which is continuously optimized around value generation.
- **Simpler performance measurement and management** SOEs have a services infrastructure which facilitates the delivery of service requests and receipt of service outputs. All measurements are done either around services delivering value. Measurements may cover
  - cost and resource consumption,
  - value generation,
  - reliability,
  - usability, and
  - accessibility.
- **Reduced gap between business and IT** A services centric approach can assist to significantly reduce the gap between business and IT. The gap is further reduced when
  - business analysts perform technology neutral business process design with sufficient semantics and rigor to facilitate clean technology mappings,
  - business analysts generate technology neutral services contracts specifications which are to be realized by IT systems,
  - the IT infrastructure is based around SOA based technologies.
- **Software as service** A SOE is naturally aligned with a business model of *software as service*, i.e. where customers do not own the software itself but pay for service usage instead.
- **Business processes visible to business** A common problem is that of business processes being largely hidden within software systems. This makes it traditionally difficult for business to perform solid business process optimization. In an SOE, business processes are designed and documented by business analysts in a technology neutral way with business being able to take full ownership of the business processes. The monitorability and auditability of business processes further improves the visibility of business processes to business.
- **Simpler ROI estimation** In a services oriented enterprise cost and value generation is measured for all services across levels of granularity. Furthermore, new new services are largely assembled form existing services sourced from within or outside the organization. This framework makes it significantly simpler to perform Return-On-Investment (ROI) estimations for new services.
- **Improved resilience** An organization which is leaner and more flexible is more resilient, i.e. better equipped to survive in an environment of changing opportunities and competition.

2.8 Risks associated with a Services Oriented Enterprise

While there are a number of benefits in basing an organizational architecture on the SOE reference architecture, one should also keep in mind the risks. In particular

- **Inhibited innovation at operational level** For an SOE the natural area of innovation is at the strategic and business process design levels, i.e. at the core business levels. Grass-roots based innovation, i.e. at the operational level, is often inhibited by the very process and contract centric approach taken at these levels.
- **Ease of introducing new services may be misleading** The ease of introducing new services may lead one to be slack on standard project management controls including business case formulation, ROI estimation, and general project control.

- **Neglecting the maintenance of the organization's assets** Another risk of a SOE is that in the context of focusing virtually exclusively on service delivery across levels of granularity, the maintenance of the organization's assets (in particular those which are not critical for the service delivery of the organization's current services profile) may be neglected.

2.9 Evolving and organization into a SOE

The process of evolving an organization into a services oriented enterprise needs to be planned and carefully implemented. The approach should rather be an iterative tha a big-bang approach. Steps typically include

1. Formulate the vision for the SOE.
2. Develop and assess a proof of concept to support the business case for SOE,
3. Communicate the vision of an SOE across the organization and start growing the culture for an SOE.
4. Specifying the infrastructure (including systems infrastructure) requirements for the SOE.
5. Develop or source the architectural skills required for putting in place and validating the core SOE infrastructure.
6. Get business analysts up to speed with solid technology neutral business process and contract specification (potentially using URDAD as analysis and design methodology) and have business analysts across the organization contribute to an organization wide business model containing the organization's business processes.
7. Define the policies for security and service publication and service governance.
8. Develop or source the skills for effective service publication and get the various organizational units to define and publish the services they offer to the organization. Entrench the culture of an SOE across the organization.
9. Integrate the various applications and systems used within the organization using the enterprise services bus (ESB).
10. Define the policies for external service provider selection and oversight.
11. Extract from the technology neutral business process designs the services contracts for external service providers, register the external service providers with the ESB and have the external services accessed through the ESB.
12. Define and implement the infrastructure for business process execution.
13. Remove the business processes from the various systems and departements and have them executed by the business process execution infrastructure.
14. Shift the focus onto continuous business process optimization including continuous assessment for"best" service providers. Remove all services from the organization which are not inline with the organization's vision and mission.
15. Evolve to dynamic service provider selection and full global integration of the enterprise.

2.10 Exercises

- Consider one of your organisation's services it offers to clients. In the context of a services catalog, list all the lower-level services (both internal and external) which are used to realise the higher-level service. For each service, list the inputs and the outputs (i.e. the exchanged value objects). Remember that services are not only offered by computer systems, but by humans and/or mechanical hardware as well.
- Discuss whether your organisation is an SOE, a traditional structure-based organisation, or something in-between. List the reasons that support your argument.

2.11 Services-Oriented Architecture

2.11.1 Introduction

SOA is a reference architecture which prescribes a *standardised infrastructure for publishing and integrating services*. It acknowledges that many services exist in different locations, accessible via different communications protocols, and which exchange different sets of information.

SOA intends to make such 'incompatible' services not only compatible, but also able to participate together in business processes which are hosted by the SOA infrastructure itself.

Though the concepts promoted by SOA is not particularly complex, a great amount of confusion exists as to what exactly SOA *is*, due to there being no official all-encompassing standards body which formally defines and evolves SOA. Instead, many industry role players have slightly differing opinions of SOA, a combination of which through we can draw our own conclusions as to the most important principles of SOA.

The key *elements* of an SOA are

- **Services** Autonomous, stateless, contract-driven service providers
- **Service Integration Infrastructure (The Enterprise Services Bus, or ESB)** An environment within which to integrate and manage heterogenous services
- **A Business Process Execution engine (BPEE)** A facility through which existing services can be orchestrated to offer a new, higher-level service to clients.

The *goals* of the Services-Oriented Architecture reference architecture are

- improved flexibility / agility
- improved integrability
- reduced time-to-market
- reduced technical skills requirement to implement business processes

Note
SOA does not itself provide the container within which services are hosted. Separate business logic containers (such as Java EE, or Microsoft .NET) continue to be used, for they posess the qualities of hosting a scalable, secure service with services such as persistence. SOA specifies the infrastructure to publish, integrate, and orchestrate services through.

2.11.2 The Principles of a Services-Oriented Architecture

2.11.2.1 Different opinions

Various prominent role players from different organisations have, at one time or another, aired their views on what the principles of SOA are. Though no single one of these sets can be considered authoritative (since there is no official standards body for SOA) they do present a useful base from which to extract a common understanding of the SOA reference architecture.

A common description that conveys the essence of Services-Oriented Architecture is

> SOA takes the existing software components residing on the network and allows them to be published, invoked and discovered by each other. SOA allows a software programmer to model programming problems in terms of services offered by components to anyone, anywhere over the network.

Nick Gall, from Gartner, defines SOA as ‘SOA is an architectural style which is’

- modular
- distributable
- described
- sharable
- loosely coupled

‘To the degree a system exhibits all five, the more it qualifies as representing the SOA style.’

Thomas Earl, in his book ‘SOA Principles of Service Design’, lists the following principles:

- Service Contracts
- Service Coupling
- Service Abstraction
- Service Reusability
- Service Autonomy
- Service Statelessness
- Service Discoverability
- Service Composability

Anne Thomas Manes, from Burton Group, states: ‘From my perspective, the overarching principle governing SOA is separation of concerns. This principle helps you determine how to factor functionality into services. Thomas Erl discusses service factoring and granularity in the SOA Fundamentals section of his book rather than treating Separation of Concerns as a principle.’

The 4 tenets of Indigo as defined by Don Box (which has been incorporated into the Microsoft tenets of SOA), state that

- Boundaries are explicit
- Services are autonomous
- Services share schema and contract, not class
- Service compatibility is determined based on policy

Finally, The 10 Principles of SOA, as expanded on the above 4 tenets, by Stefan Tilkov, state that

- Explicit boundaries
- Shared contract and schema, not class
- Policy-driven
- Autonomous
- Wire formats, not programming language APIs
- Document-oriented
- Loosely coupled
- Standards-compliant
- Vendor-independent
- Metadata-driven

2.11.3 Problems SOA attempts to solve

Consider the order processing workflow shown in the following figure:

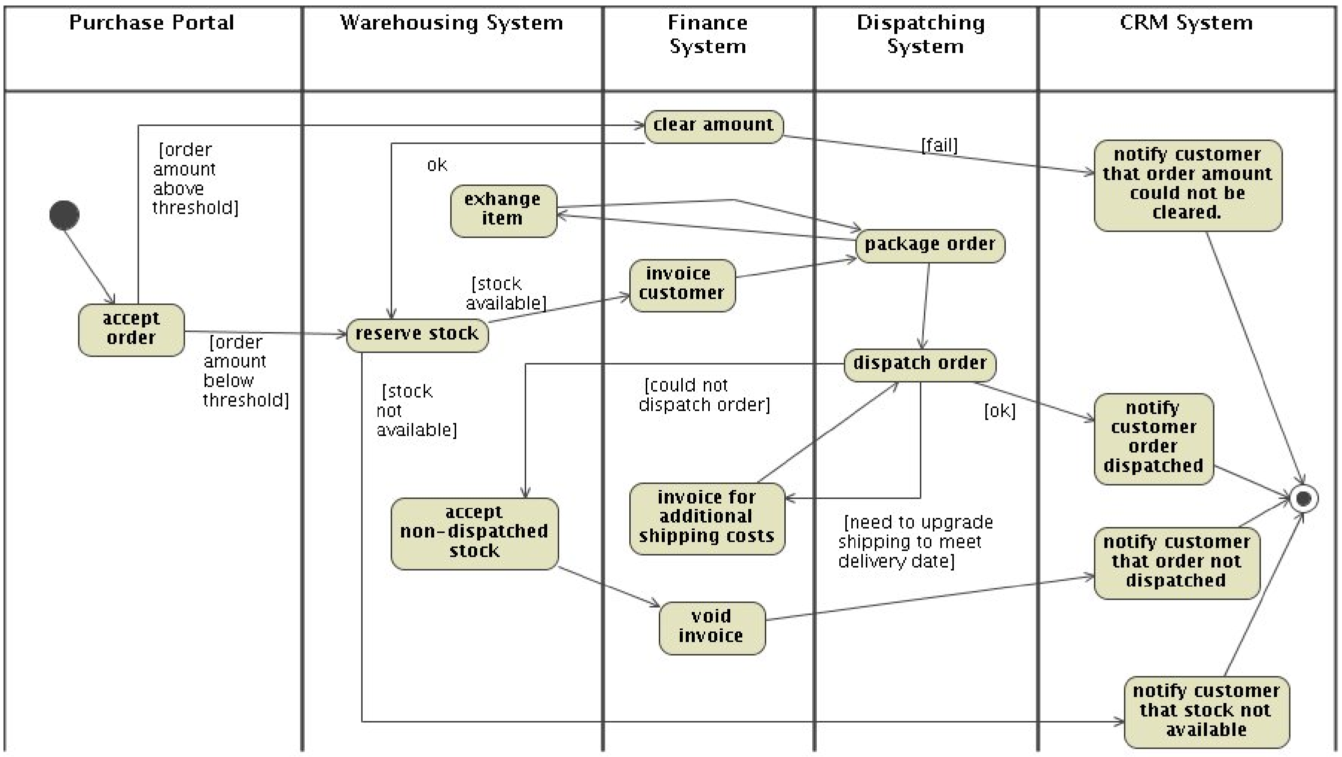

Purchase Portal
Warehousing System
Finance System
Dispatching System
CRM System
accept order
[order amount above threshold]
[order amount below threshold]
clear amount
ok
[fail]
notify customer that order amount could not be cleared.
exhange item
package order
invoice customer
reserve stock
[stock available]
[stock not available]
dispatch order
[could not dispatch order]
[ok]
notify customer order dispatched
invoice for additional shipping costs
[need to upgrade shipping to meet delivery date]
accept non-dispatched stock
notify customer that order not dispatched
void invoice
notify customer that stock not available

Figure 2.2: An (admittedly convoluted) order processing workflow

A direct implementation of that workflow would result in a collaboration context with a large number of dependencies between the systems:

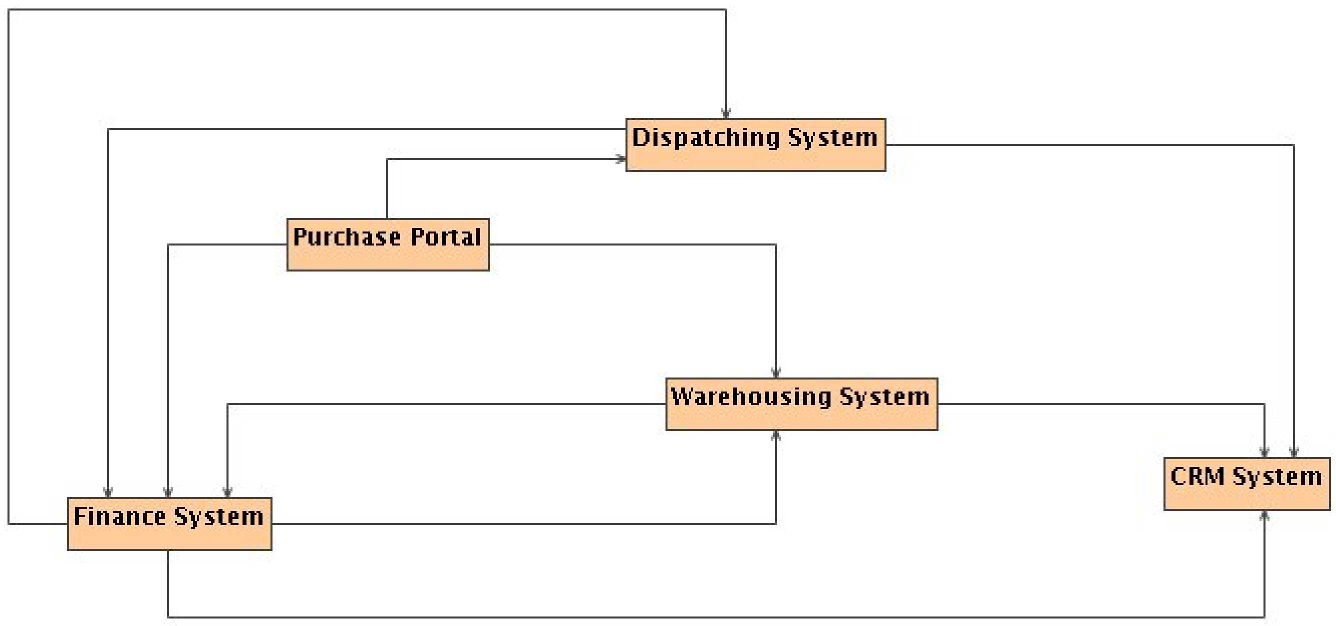


Figure 2.3: An excerpt of the collaboration context showing message paths required for the direct-integration order processing workflow

2.11.3.1 Problem 1: Integration costs

Out of the 20 possible message paths between the 5 systems, we require 10. Thus, in order to realize this single use case we require 10 adapters between 5 systems. Additional use cases would typically increase the number of adapters required.

Large organizations deploy many business processes realizing a wide range of use cases across their systems. The number of systems is often significantly larger than. For full interconnectivity one would require `N^2 - N` adapters (for 12 systems this would imply 132 adapters). In practice one would typically not require *full* connectivity, but rather approximately half of the adapters.

2.11.3.2 Problem 2: Business process maintainability

If we look at the above example then you will realize that the business process specification is not localised in any of the system, but that the workflow is implicitly encoded across the various systems.

This has a direct negative impact on maintainability of the business process.

2.11.4 Services

2.11.4.1 Services offer value to clients

A service exists to fulfil a contract. The contract exists to express the requirements (use-case) of a particular stakeholder in a system. To this end, services are best implemented as a direct mapping from a technology-neutral model which was constructed using a process such as URDAD, which results in each contract - at each level of granularity in the system - being designed in exactly the same (client-centric) manner.

2.11.4.2 Services are contract-driven

The behaviour of a service should always satisfy the constraints of the *contract* (service description) it realises. The contract specifies the operations (use cases), the pre- and post-conditions of each operation, as well as any quality requirements around these (security, reliability, performance, etc).

By forcing a service's implementation details ot be hidden behind a well-defined contract, SOA fosters test-ability, as well as plug-ability of different service providers realising the same contract.

In SOA a single, standard, contract language is enforced across services, even though each service may be *realised* in a different implementation technology.

2.11.4.3 Services are autonomous

The principle of service autonomy (which is more of a mindset than an enforceable technicality) simply means that each service was designed and implemented without a technical concern for other (perhaps existing) services, and that services should not hold influence over one another. In the organisation, different role players (departments) may expose different services to a common services infrastructure, and the implementation of these services do not need to (and should not) be concerned with adhering to common contracts, storing information in a common database, or using a common programming language.

Such decisions are not driven by SOA principles, but purely by the practicalities of the organisation, i.e. the available skills for development and administration.

2.11.4.4 Services are stateless

Services should not (and cannot) maintain *conversational state* with clients. This forces the messages that are exchanged to contain all required information in order for the service provider to sensibly deduce the business context within which it is made. This is, in anyway, a good practise to adhere to, even in stateful object-oriented technologies.

For example, the following exchange (in english) maintains conversational state:

```
CLIENT:         Please repair my car. It's the Blue Porsche (registration
                number ABC123GP) parked in front of the workshop.

REPAIR CENTRE: We've assessed the damage and estimated cost, have a look at
                this quote (ref no 8872) and let us know if you accept
                it (in order for us to proceed).

CLIENT:         Looks good to me, go ahead.

REPAIR CENTRE: Excellent. We'll let you know when your car is done, sir.
```

The problem with this exchange is that, because conversational state is maintained, the idea of *object identity* is important: If the first message exchange occurred physically at the workshop, and the second exchange occurred a day later via telephone, the telephone operator will (in all likelihood) not be able to understand the message from the client:

```
CLIENT:         Please repair my car. It's the Blue Porsche (registration
                number ABC123GP) parked in front of the workshop.

REPAIR CENTRE: We've assessed the damage and estimated cost, have a look at
                this quote (ref no 8872) and let us know if you accept
                it (in order for us to proceed).

(one day later, the telephone rings)

REPAIR CENTRE: Good afternoon, how may I help you?

CLIENT:         Looks good to me, go ahead.

REPAIR CENTRE: Huh?
```

The following version of the conversation has been rewritten in a *stateless* manner, which means that it does not depend on the object identity of the service provider, nor on conversational state:

```
CLIENT:        Please repair my car. It’s the Blue Porsche (registration
               number ABC123GP) parked in front of the workshop.

REPAIR CENTRE: We’ve assessed the damage and estimated cost, have a look at
               this quote (ref no 8872) and let us know if you accept
               it (in order for us to proceed).

CLIENT:        I would like to confirm quote no 8872, please proceed
               with the repairs.

REPAIR CENTRE: Excellent. We’ll let you know when your car is done, sir.
```

Stateless services are an important enabler of

- freedom from communication channel (interactions pertaining to the same business process may happen through different channels)
- freedom from environmentally-imposed timing constraints, such as network time-outs or the duration of telephone calls
- increased scalability, by allowing the service provider to pool or re-use service provider objects internally in a simple way
- simpler implementation of service providers, which do not themselves ave to maintain the conversational state they have with all of their clients
- easy re-usability and test-ability: the pre- and post-conditions of services can be clearly defined, as they do not refer to past or future conversations (something which is difficult to define and test formally)

2.11.4.5 Services are composable

Services do not specify, or limit, the *context* within which they are used. This, together with a standard language for specifying service contracts, means that services can easily be re-used in the context of higher-level, composite services.

SOA prescribes a business process execution engine, where potentially long-lived business processes are assembled from individual services. This, again, is exposed as a service - ensuring that it itself could be re-used in the context of other, even higher-level services.

2.11.4.6 Services are discoverable

Information about a service (the service contract, access channels, endpoint locations) must be published through a mechanism which allows potential clients to discover the service. The actual mechanism used ma vary, and most component technologies (EJB, .NET) do have some form of a naming service through which the available services and their contracts can be discovered.

2.11.5 The Enterprise Services Bus

2.11.5.1 Introduction

Most organisations need to be able to define business processes which are realised across the various work units available within the organisation and its business partners. These work units may be systems implemented in a wide range of technologies, running in disparate physical locations.

2.11.5.1.1 What is an ESB?

An *Enterprise Service Bus* (ESB) provides a realisation of a services-oriented architecture. It is an enterprise platform which implements standard interfaces for

- communication (messaging),
- connectivity,
- transformation,
- portability and
- security.

The ESB aims to providw a *lightweight, flexible services bus* in order to reduce time-to-market. *Point-to-point integration is eliminated*, since all integration is through the bus.

2.11.5.1.2 Service orchestration

A service may be composed of a number of service steps which are themselves services. As such a services oriented architecture is largely based on the *pipes and filters* architectural pattern.

2.11.5.2 ESB Architecture

An ESB provides a light-weight architecture for deploying and integrating services across an organization.

2.11.5.2.1 The services bus

An ESB architecture revolves around a services bus, to which services are published (via their contracts). These services may either be local to the infrastructure, or remote (such as hosted by a business partner) - both cases are treated identically by the ESB. The services bus is responsible for routing messages between these services.

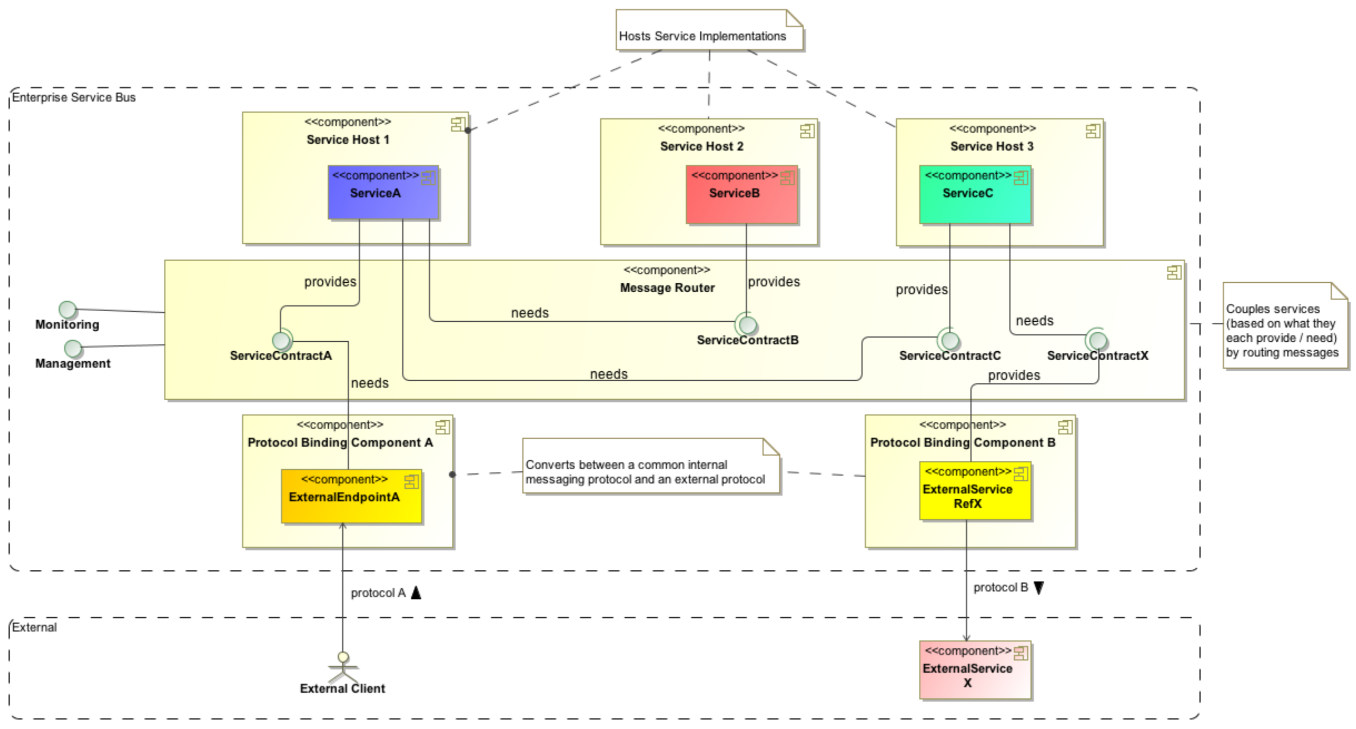

Hosts Service Implementations
Enterprise Service Bus
<<component>> Service Host 1
<<component>> ServiceA
<<component>> Service Host 2
<<component>> ServiceB
<<component>> Service Host 3
<<component>> ServiceC
<<component>> Message Router
provides
needs
provides
provides
needs
needs
provides
needs
Monitoring
Management
ServiceContractA
ServiceContractB
ServiceContractC
ServiceContractX
Couples services (based on what they each provide / need) by routing messages
<<component>> Protocol Binding Component A
<<component>> ExternalEndpointA
Converts between a common internal messaging protocol and an external protocol
<<component>> Protocol Binding Component B
<<component>> ExternalService RefX
protocol A
protocol B
External
External Client
<<component>> ExternalService X

Figure 2.4: An ESB architecture revolves around a services bus

2.11.5.2.2 Distributed services architecture

The ESB as single entity is purely virtual: it could span the organization, with services deployed across a potentially large number of co-operating hosts.

2.11.5.2.2.1 Service containers

Nodes on the services bus host service containers (or service hosts), and services are deployed to these. As soon as a service is deployed in a container, it is immediately *available across the services bus*. Service containers may either host the services themselves, or merely proxies to external services.

2.11.5.2.2.2 Managing service containers

Service containers are usually managed through JMX (the Java Management Extension API) and/or other management tools which may be specific to the ESB provider.

2.11.5.2.2.3 Service replication

A service can be deployed across a number of services containers, i.e. across a number of physical nodes. This architecture aims to facilitate saleability and reliability (fail-over safety).

2.11.5.3 What services do the ESB provide ?

The ESB, as integration infrastructure, provides a number of key services which enable Services-Oriented Integration across services.

2.11.5.3.1 Indirect service requests

If somebody wishes to call a service published on the ESB, it does not send a message directly to the service. Instead, it requests it from the ESB directly, which will route the request to, and the response from, the service in question.

Service clients could request services *abstractly* (i.e. only based on the contract which is implemented, not caring "who" services the request) or *concretely* based on a specific service endpoint (by name).

2.11.5.3.2 Service coupling

Every service deployment to a service container must indicate (via contracts) both the services it *provides* to the environment, as well as the services it *needs* from the environment.

Based on this information, the ESB can work out the message routing between services. This powerful feature completely de-couples services from one another, guaranteeing service autonomy and plug-ability.

2.11.5.3.3 Message delivery

The ESB is responsible to deliver messages, in accordance with the message exchange pattern(s) specified in the contract, to services. Most ESBs allow one to specify the quality trade-offs for message delivery: Higher performance with potential failure if the ESB goes down mid-call, vs. reliable message delivery via persistence and acknowledgement or receipt.

2.11.5.3.4 Message transformation

More often than not, request messages to services need to be constructed by transforming existing messages. This is especially common when incompatible service contracts exist, and an *adaptor* needs to be built.

Since most ESBs use XML as the internal messaging format, they typically allow the deployment of XSLT-based transformation services to an XSLT service container, which is then called just like any other service.

2.11.5.3.5 Service access via diverse protocols

The ESB must support a range of protocol binding components, which each support a particular communication protocol, such as SOAP/HTTP, SMTP (e-mail), CORBA, JMS, CICS, and so on. Each binding component can be configured to either *expose a service endpoint* to external clients, or host a *proxy to an external service*.

This is implement no different to the services hosted in the ESB's service containers: Each binding component deployment either *provides* or *needs* (consumes) a specific service contract. This, coupled with the ESB's service coupling featured (based on the contracts) makes it possible to expose/consume any service via almost any protocol, without the service's knowledge thereof.

2.11.5.3.6 Service Monitoring and Management

The ESB must expose services to

- **Control the life cycle of services and assemblies** It must be possible to deploy, undeploy, start, suspend and resume individual services, as well as entire groups of services which form part of a particular integration assembly
- **Configure the message delivery channels** It must be possible to configure the reliability/performance characteristics, as well as e.g. the storage of messages being delivered by the messaging infrastructure.
- **Optimise connection, thread- and object pools** This is useful to tune the ESB to deal with varying different loads.

2.11.5.3.7 Security

As an integration mechanism, the ESB does not typically enforce authorisation itself: It is merely responsible for passing along any authentication metadata collected by protocol binding components to services.

2.11.5.3.8 Transaction management

Services are often realised over a number of lower-level services which address specific aspects of that service. Often a service needs to be completed as a whole or otherwise the service steps need to be undone. To this end a SOA implementation needs to support transactions.

2.11.5.4 Roles

A number of common roles have emerged in the SOA space, indicating the responsibilities of certain individuals. Having a common language to describe these roles (just like with Java EE) allows the industry to communicate the intentions around tools, training courses, employee resumé requirements, etc. The following roles are pertinent:

2.11.5.4.1 Business-logic developer

A business logic developer writes the individual service implementations without worrying about

- where the service is to be deployed,
- where the services it depends on are deployed,
- non-functional requirements such as security, reliability

2.11.5.4.2 Service orchestrator

Service orchestrators

- integrate services into higher level services and
- specify transactional requirements across the resultant work flow.

2.11.5.4.3 Service deployers

Service deployers

- decide on where services are deployed,
- register services with the registry,
- and assign authorization requirements to services.

2.11.5.4.4 Administrators

Administrators are responsible for ensuring that the

- nodes hosting services are available,
- the ESB infrastructure is available, and
- system load is manageable (i.e. performance is adequate)

2.11.6 The Business Process Execution Engine

A critical part of any SOA implementation is the support for a *business process execution engine*, whereby the infrastructure itself is responsible for executing and managing business processes which have been assembled across services.

Though most service implementation technologies (such as Enterprise JavaBeans) have the ability to define workflows across lower-level service providers, what sets an SOA Business Process Execution Engine apart is

- the declarative nature of the process
- the support for long-running business processes
- the complete de-coupling of the business process from the low-level services via the ESB

2.11.6.1 Declarative business processes

Instead of being implemented in a compiled language such as Java, most BPEEs support business processes written in a declarative way (typically using an XML vocabulary). Such languages are usually simpler than fully-fledged programming languages, concentrating instead on the tasks typical of service orchestration, such as

- invoking services via the ESB,
- transforming request and result messages,
- performing conditional and/or repeated workflow steps
- performing tasks in sequence or in parallel
- managing faults / exceptions and performing compensating actions for failed actions

Because of the declarative, XML-based nature of such languages, intuitive graphical tools are made possible to allow persons with a weaker level of technical skill to assemble business processes from existing services. Though such tools are in theory possible for any programming language, the simpler and purpose-designed nature of business process languages make such tools far easier to implement.

2.11.6.2 Long-running business processes

Most services offered by typical object-oriented and client/server systems are of a simple synchronous (request-response) nature. Furthermore, it is assumed that such services will complete in a very short time (usually a second or less).

Though the concept of a long-running business process is a natural one in the business world, traditional systems do not explicitly support this in any way - it is up to the developers to ‘tie together’ several different service interactions into a ‘business process’, usually requiring a lot of manual work around storing and modifying the *state* of the business process.

In SOA, the BPEE on the other hand allows the implementor to specify one (potentially long-lived) business process, as well as all the (either synchronous or asynchronous) interactions between the process, its clients, and it’s lower-level service providers, which could occur during the lifespan of the business process.

The BPEE creates and manages *instances* of such business processes on demand, completely freeing the implementor from the rigours of reliably maintaining the state of a business process which could last anything from milliseconds to years.

2.11.6.3 Removing processes from the hands of people

A surprising number of organisations - even ones with mature and well-implemented information systems - have solid low-level services, but with the actual business process being *implicitly* applied across these services by human beings. Unless the on-the-fly ingenuity and adaptability of this implementation technology is required by business, most typical organisations could greatly benefit from having one well-defined, executable ‘version’ of the business process managed by the SOA infrastructure, with the human beings rather playing the role of lower-level service providers as and when needed.

2.11.7 An Overview of SOA technologies

SOA abstractly specifies a number of conceptual components, such as *services*, an *enterprise services bus* for service integration, as well as a *business process execution engine* to host business processes. In this section, we introduce some of the common technologies used today to realise these components.

2.11.7.1 Services contracts

As SOA technologies have universally settled on the *web services* technologies when it comes to service interaction / messaging, it is the associated WSDL (Web Services Definition Language) which is used to specify services contracts.

WSDL is a general container format for expressing any of

- the structure of exchanged value objects (using XML Schema),
- the functional requirements around a service, such as the sequence of exchanged messages and the preconditions,
- the non-functional requirements around a service, such as the security and performance requirements, using WS-Policy and WS-PolicyAttachment annotations,
- how a particular service is bound to a particular transport protocol (such as SOAP/HTTP), and
- concrete service endpoints (such as URLs where messages may be sent to)

The original WSDL 1.1 standard has been superseded by the WSDL 2.0 standard, although most SOA implementations will support both for the foreseeable future.

2.11.7.2 Service implementations

SOA in principle does not make any statements as to the physical realisation of services, as it is concerned only with the integration and orchestration of such services. When deciding to implement a service, it should always be hosted in an environment which was chosen because it meets the *qualities* required by the contract, such as performance, scalability, and so on.

Several suitable enterprise component technologies (most of which pre-date SOA) exist to host general business logic services, such as Enterprise JavaBeans. If fewer qualities are required, a more lightweight approach may be taken, such as hosting Java objects in a Spring Framework or POJO container. Most of these environments will provide access to general services such as persistence to relational database via an object-relational mapper.

If the purpose of a service is simply to support integration (such as transforming messages between services) then it may be better implemented by hosting one or more XSLT transformations in an XSLT container (usually provided by any SOA environment).

Of course, if a service implementation requires, as qualities, a great deal of innovation and problem-solving ability, it may of course be implemented by a human which has been trained for the job.

In *all* of these cases, a suitable adaptor needs to exist in order to plug the service runtime into the SOA runtime, in order for message exchange to take place. These are usually standardised (and freely available) for a great number of implementation technologies.

2.11.7.3 Integration infrastructure

As the heart of the SOA infrastructure, any standards around the Enterprise Service Bus are sure to dictate the entire systems development, packaging and testing process. Public standards in this area allow organisations to use ESB frameworks without becoming vendor-locked to a particular framework.

Not being vendor-locked is especially important during the development process, as the potential complexity of assembling composite integration scenarios may require a number of sophisticated tools. It is unlikely that a single vendor can provide all the tools to keep all the users of a particular ESB productive.

An ESB standard should cover not only the service hosting and message routing, but preferably also standardise the deployment, management and monitoring process.

The primary standard for ESBs is JBI, *Java Business Integration*. It specifies standards not only for the inner workings of the ESB (such as message routing and a normalised message format) but also for the contract between the ESB and components plugged into it (such as a Business Process Execution Engine) as well as standardised management and monitoring using JMX (Java Management eXtensions). It goes as far as to standardise tasks for the Apache Ant build tool, which enables development projects to manage and deploy components identically regardless of JBI implementation.

An ecosystem of JBI components (Service Engines, Protocol Binding Components) have also come into existence, which means that one no longer has to choose an ESB based on the components they provide: With JBI, one ESB's components may be installed in another ESB. Examples of JBI implementations are Apache ServiceMix, OpenESB and Petals ESB. JBI is currently still quite immature (version 1.0) which means that many practical aspects of the ESB are, in fact, not yet governed by the specification. JBI 2.0 aims to practically simplify ESB development in a similar manner to how EJB 3.0 greatly simplified Enterprise Java development without changing the fundamental (and sound) architecture.

An alternative SOA solution architecture, SCA (Service Components Architecture) is also promoted, especially by IBM and BEA Systems. This proposes a different, de-centralised infrastructure not based on the *mediated message exchange* paradigm of JBI. While JBI is a specification for ESBs implemented using Java, SCA aims to be more technology neutral, applying a common language runtime concept similar to Microsoft's .NET.

2.11.7.4 Service clients

Ultimately, the services published via the SOA infrastructure needs to be made use of by clients, if they are to have any value. By virtue of the nature of SOA, most service clients are in fact simply other services. It is, however, often the high-level services accessed by humans which carry the highest profile, due to their visibility.

Since and SOA environment contains adaptors to most communications- and messaging-technologies, many external service clients could simply access the service using their protocol of choice, as long as the correct messages are sent. Most external clients will access services via SOAP web services, and for these, standard web services-to-object frameworks could be used, such as JAX-WS for Java clients. Such frameworks can generate all required artifacts (service stubs, exchanged value objects, etc) by examining the *service contract* published by the services infrastructure.

When one considers graphical user interfaces (and, specifically, web-based user interfaces) it is likely that SOA will, over time, bring forth a big change in the way that these are implemented. Almost all current user-interface technologies contain a control layer, which actually hosts the application work-flow at the topmost level of granularity. Examples include Java Server Faces, Jakarta Struts, and Microsoft ASP.NET. Since SOA aims to move the business process to the ESB itself (via a Business Process Execution Engine) we believe there will be an emphasis on more modern, lightweight user interface technologies. All that is fundamentally required is the ability to interactively populate a service's request messages, and display the response messages to the user. An example of such a technology is W3C XForms, which replaces standard web forms in the XHTML version 2.0 standard for web pages.

2.11.7.5 Service registries

When discussing the publication of a service (via its contract) in a services registry, one must consider the separate cases of publishing the service *to the services infrastructure* (i.e. in order for other services in the same SOA infrastructure to find and use), as well as publishing the service *to external clients* (i.e. in a public marketplace in order for prospective clients ad business partners to discover).

Since the core responsibility of an SOA is the integration and orchestration of services, most implementation frameworks include a well-defined, query-able services registry. Service endpoints are typically published automatically as they come online. This holds true for all JBI-based ESBs.

The two prominent standards for public registries are UDDI and ebXML. Unfortunately, it appears as it they are currently not universally adopted, nor do SOA frameworks commonly support automatic publication to such external registries. Publishing a service to an external, public registry is currently thus a manual and explicit business activity.

2.11.7.6 Business Process Execution Engine

The primary standard for declaratively specifying potentially long-running business processes is WS-BPEL, the *Web Services Business Process Execution Language*. WS-BPEL is managed by the OASIS standards organisation, and aims to be a robust language for managing an orchestration of lower-level services, itself again exposed as a service.

Though SOA dictates a business process execution engine ‘provided by the ESB infrastructure’, in practise almost no ESB can natively execute business processes. Instead, a WS-BPEL process is usually deployed to a service engine hosted by the ESB.

An example of a BPEE is Apache ODE (Orchestration Director Engine), a standard JBI component which may be installed to any JBI-compliant ESB.

Note
Because of the practical issues often encountered with immature technologies such as WS-BPEL, standard programming languages like Java is also very commonly used for business process execution.

2.11.8 Benefits of SOA-based software systems

SOA based systems are maturing and most large organizations are either exploring their use or are actively moving to using them. This obviously shows that IT teams and organizations expect to obtain significant benefits from these technologies.

2.11.8.1 Reduced complexity

The shift from a structured to a services centered focus results in a significant reduction of conceptual and technical complexity. At any stage one typically needs to concern oneself only about a particular service at a particular level of granularity. The higher level services and lower level service implementations are decoupled through well defined services contracts.

2.11.8.2 Shorter time to market and increased flexibility

A SOA typically enables one to achieve shorter time to market and increased flexibility through the complexity reduction, increased level of reuse which one typically achieves within this infrastructure, and through the simpler integration environment.

2.11.8.3 Simpler, more reliable paradigm for reuse

Published services with their associated services contracts nurtures an environment of reuse.

2.11.8.4 Testability

Well defined services contracts directly lead to testability.

2.11.8.5 Standards based

SOA standardizes many aspects from discovery to communication, addressing, contract specification, data structure and data transformation specification, security, messaging and business process specification to service management and other aspects. These standards

- reduce integration costs,
- secure long-term investment,
- are often more mature than rapidly changing proprietary solutions,
- have usually good tools support,
- reduce vendor lock-in and
- reduce technology and methodology proliferation.

2.11.8.6 Less implementation errors

A SOA typically enables one to reduce the bug ratio through the contracts based approach, testability, reduced complexity and increased reuse.

2.11.8.7 Monitorability and manageability through the framework

Monitoring and management facilities do not have to be built into applications, they are provided by the SOA infrastructure itself.

2.11.9 Risks around SOA-based software systems

A number of SOA projects have failed. This points to that there are still real risks associated with SOA based technologies.

2.11.9.1 Vendor lock-in

Due to an incomplete set of public standards, standards which are not yet ratified and insufficient standards support, or due vendor provided convenience tools which encourage the use of vendor specific features, SOA implementations often still end up having significant vendor lock-in.

This typically results in increased porting and production costs, reduced skills pools, and increased risk.

2.11.9.2 Performance

Naive usage of SOA based technologies for performance-critical work may result in performance issues associated with marshalling/demarshalling and messaging.

2.11.9.3 Immature technologies

Many of the SOA frameworks are still immature and some of the standards are still in draft. This results in tools problems and incomplete feature sets.

2.11.9.4 Technology abuse

SOA technologies are still relatively novel and often there is insufficient conceptual or technical understanding in these technologies. This may result in using the technology incorrectly and not obtaining the envisaged benefits.

2.11.9.5 Non-business-driven projects

One of the core benefits of using SOA based architectures is that it has the ability to reduce the gap between business and IT, with business and IT communicating with similar languages. The core benefit from this is only realized if SOA projects are business driven (i.e. if realised within an Services-Oriented Enterprise).

Part II

SOA in practice

Chapter 3

Developing Services

3.1 Writing a service contract

3.1.1 A revision of design-by-contract

3.1.1.1 Design By Contract

Design by Contract is a design technique introduced in 1986 by Bertrand Meyer for developing quality software based on formal contractual requirements which are testable and validated in code. Bertrand Meyer went on to make design by contract an integral part of the Eiffel programming language.

3.1.1.1.1 Applicability of Design by Contract

Even though *Design by Contract* was originally a concept developed for software design, it is now commonly applied across various design fields covering software, hardware and business process design.

3.1.1.1.2 Quality and productivity

K. Fujino's quote,

> *when quality is pursued, productivity follows*

has become a widely accepted mantra.
In other words, it is been proven unwise to believe that

- your deadlines are too pressing to follow a quality-driven approach like design-by-contract,

but rather that

- you cannot afford *not* to follow a quality driven approach with pressing deadlines.

Most enterprise reference architectures, be it a component infrastructure like Enterprise JavaBeans, or an integration infrastructure like Services-Oriented Architecture, in fact *enforce* a design-by-contract approach.

3.1.1.1.3 Pre-conditions, post-conditions and invariants

Design by contract is based on the premise that the functional requirements for a service provider can be specified by defining the interface for the service provider together with pre-and post conditions for each service, and optionally invariance constraints for the service provider as a whole.

1. **Preconditions** The preconditions are those conditions under which a service provider may refuse the service without breaking the contract.

   Note
   Service providers would typically raise a signal (throw an exception) to notify the client that a requested service is not being provided because some pre-condition is not met.

2. **Post-conditions** The post conditions are those conditions which must hold after successful realization of the service.
3. **Invariance constraints** The invariance constraints are rules that must always hold for the object (providing the service) to be in a valid state -- it must be satisfied before the service is requested as well as after the service has been rendered or refused.

3.1.1.1.4 Example: Account

As an example, consider an account. The invariance constraints which must apply for the account to be in a valid state are that the sum of the credits minus the sum of the debits must always yield the account's balance. A precondition for the debit service is that there are sufficient funds -- if there are insufficient funds the account may refuse the service without breaking the contract. Post conditions for the debit service may be that the amount is subtracted from the balance and that the transaction has been inserted into the transaction history. The credit service, on the other hand, needs to be provided unconditionally while the post conditions of the credit service are similar to those of the debit service, just that the amount must be added to the balance.

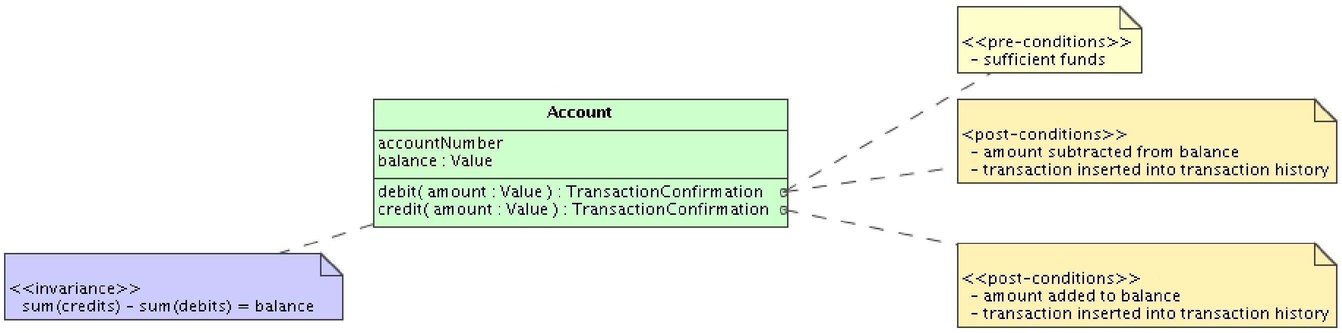

Account
accountNumber
balance : Value
debit(amount : Value) : TransactionConfirmation
credit(amount : Value) : TransactionConfirmation
<<pre-conditions>>
- sufficient funds
<<post-conditions>>
- amount subtracted from balance
- transaction inserted into transaction history
<<post-conditions>>
- amount added to balance
- transaction inserted into transaction history
<<invariance>>
sum(credits) - sum(debits) = balance

Figure 3.1: Pre- and post-conditions and invariance constraints for an account.

3.1.1.1.5 Exceptions versus errors

An exception is raised when a service provider refuses a service due to a *precondition not being met*. This is a scenario where the service provider may refuse the service without breaking the contract. It does thus not present an error - the system is not in an inconsistent state, and it may continue operating normally.

If, on the other hand, all preconditions are met and the service provider is still unable to realise the service such that all post-conditions will be met and that no invariance constraints will be violated, then the server may want to acknowledge to the client that the contractual obligations cannot be met by raising an error.

3.1.1.1.6 Design by contract and specialisation

When, for example, defining an Account sub-class such as *CreditCardAccount* (which defines a different business process for the `debit` service) one needs to take into account the following:

- Instances of sub-class must provide the service under no more preconditions than the super-class, i.e. the service may not be refused for any other reasons than that there are insufficient funds, though insufficient funds itself may arise under different conditions (the credit card could allow for a negative balance while normal accounts could potentially not allow this).
- The post-conditions for account's debit service must still apply to that of credit card accounts, though credit card accounts could add additional post-conditions like that an associated voyager mile account must be credited with a certain amount of voyager miles.
- The invariance constraints for accounts must still hold for credit card accounts, though credit card accounts could add further invariance constraints around the additional state maintained for credit card accounts.

Caution
Not adhering to any of these design by contract rules would directly violate substitutability, i.e. one would no longer be able to substitute an instance of the sub-class for an instance of the more generic class.

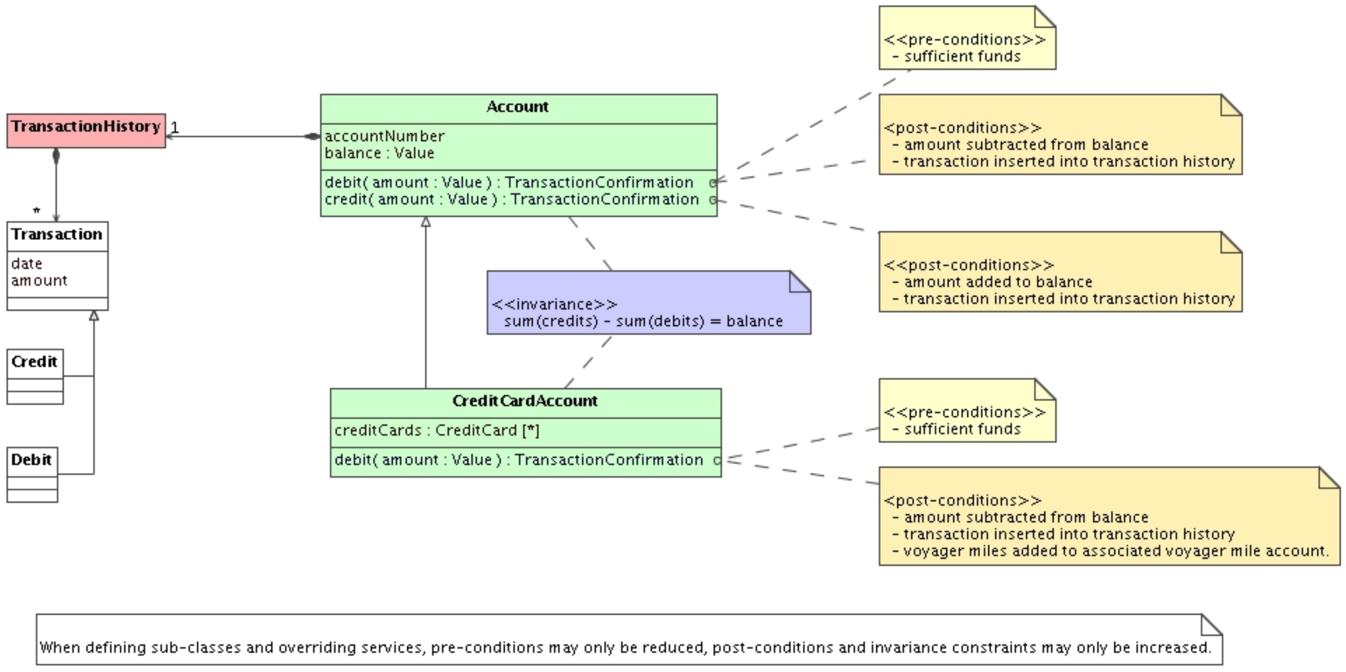

<<pre-conditions>>
- sufficient funds
TransactionHistory
1
Account
accountNumber
balance : Value
debit(amount : Value) : TransactionConfirmation
credit(amount : Value) : TransactionConfirmation
<post-conditions>>
- amount subtracted from balance
- transaction inserted into transaction history
*
Transaction
date
amount
<<post-conditions>>
- amount added to balance
- transaction inserted into transaction history
<<invariance>>
sum(credits) - sum(debits) = balance
Credit
CreditCardAccount
creditCards : CreditCard [*]
debit(amount : Value) : TransactionConfirmation
<<pre-conditions>>
- sufficient funds
Debit
<post-conditions>>
- amount subtracted from balance
- transaction inserted into transaction history
- voyager miles added to associated voyager mile account.
When defining sub-classes and overriding services, pre-conditions may only be reduced, post-conditions and invariance constraints may only be increased.

Figure 3.2: Applying design by contract rules for specialisation.

3.1.1.1.7 Design by contract and implementing interfaces

We can assign pre- and post-conditions to a service specified in an interface, and invariance constraints to the interface as a whole. This would require that any implementation of that interface would have to

- provide each service, subject to no more pre-conditions than what is specified in the interface,
- realise each service such that all the post-conditions are met once the service has been provided, and

- ensure that the published state of the implementing object never violates the invariance constraints.

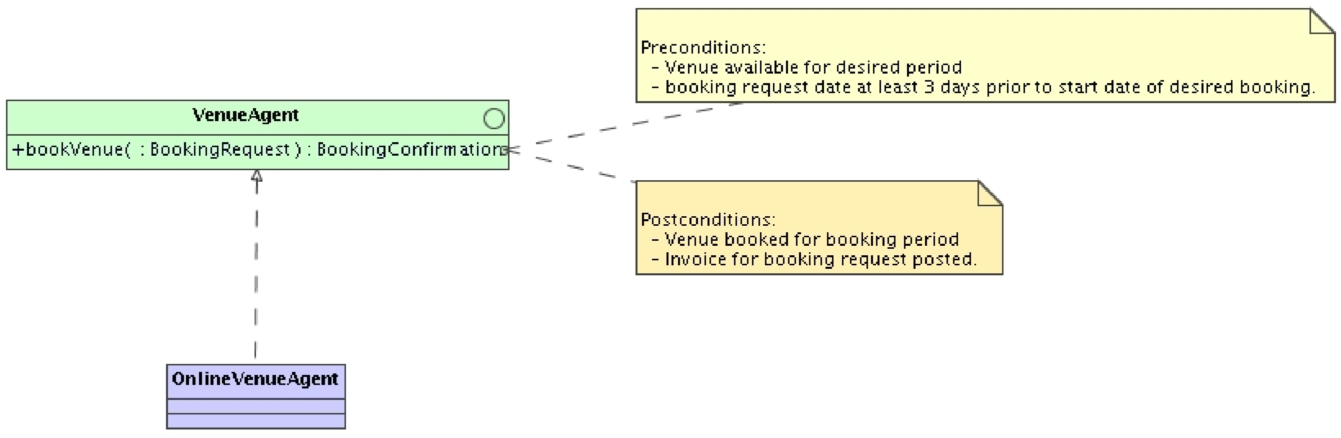

VenueAgent
+bookVenue(: BookingRequest) : BookingConfirmation
Preconditions:
- Venue available for desired period
- booking request date at least 3 days prior to start date of desired booking.
Postconditions:
- Venue booked for booking period
- Invoice for booking request posted.
OnlineVenueAgent

Figure 3.3:

Very often one uses an interface with pre- and post-conditions on the services to specify the service requirements for service providers without constraining the state of the service provider. In such cases there would be no invariance constraints. The previous figure shows an interface with pre- and post-conditions specified for the required service. Any particular implementation of a venue agent must be assigned a business process which refuses the service under no more conditions than what is allowed via the pre-conditions and which realizes the service in such a way that the post conditions are met for any success scenario (i.e. any scenario where the service was not refused due to some pre-condition not having been met).

3.1.1.1.8 Contracts

An interface together with the pre- and post-conditions on the services and potentially some invariance constraints around the published state goes a long way towards defining a complete contract facilitating full pluggability of service providers. To complete the contract and allow for full pluggability of any service provider realizing the contract we still need to add

- the class diagrams for the value objects exchanged, and
- the quality requirements for the service provider.

3.1.1.1.8.1 Quality requirements

Having specified that a venue agent needs to be able to process booking requests and having associated the pre- and post-conditions for the required *bookVenue* service will not yet guarantee us that any service provider realizing the interface subject to these design-by-contract constraints will indeed be pluggable. We may still have certain quality requirements for the services. For example, our business processes may require that we receive the booking information within an hour of placing the booking request. Some venue agents which implement the above interface may not be able to adhere to this *performance* requirement and may thus not be pluggable.

Another quality requirements may specify a certain *scalability*, the ability to process a certain number of booking requests per unit time (e.g. per day).

3.1.1.1.8.2 Services without pre- and/or post conditions

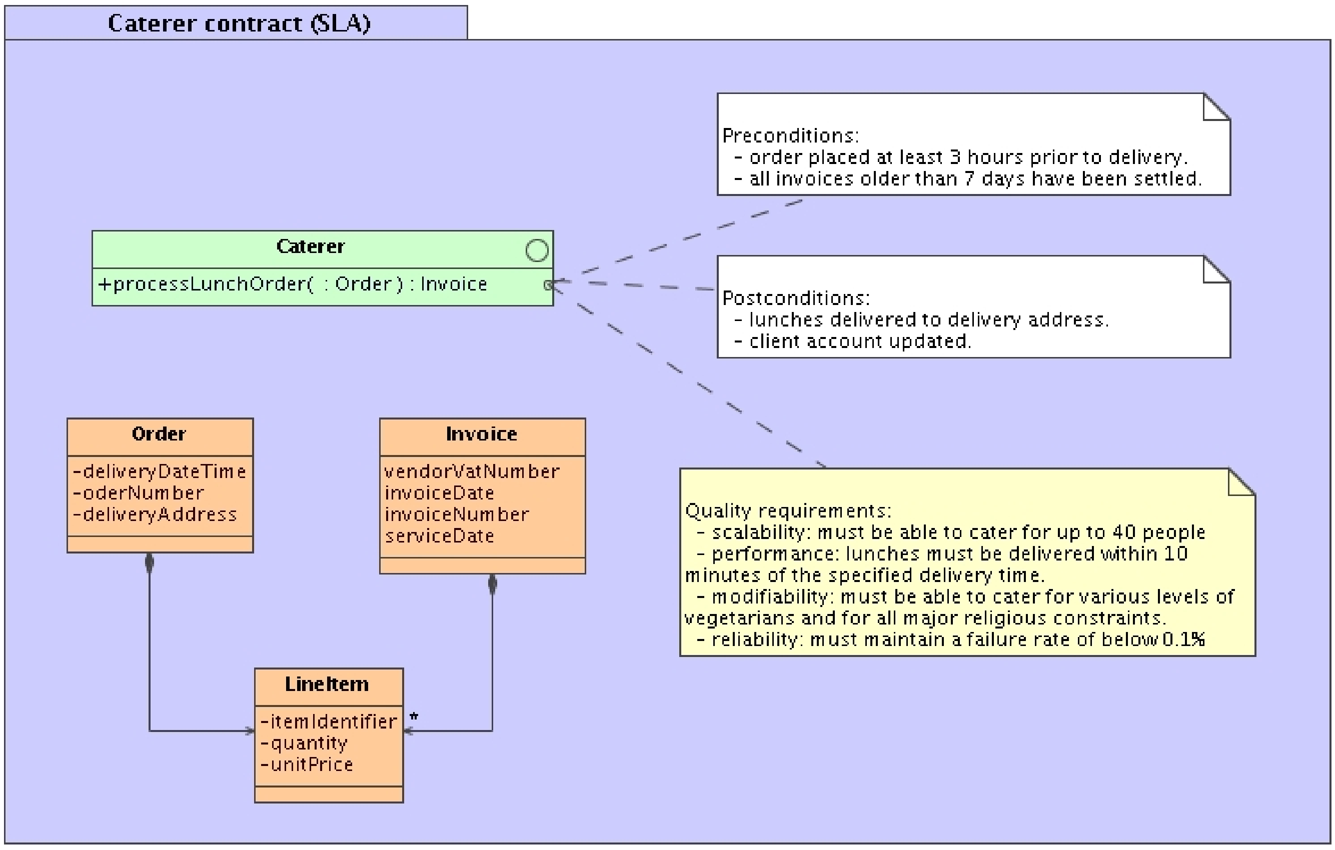
Some services may need to be provided unconditionally. For such services one would not specify any preconditions. Similarly, some services would return some result, but would have no further remnants after the service has been provided. Such services would have no post-conditions.

Consider the *getTime* of a *Clock*. This service should always be provided, i.e. if, under any conditions, the clock does not provide the service, then there would be an error with the clock. Furthermore, the clock will return the current time, but there would be no further persistent remnants of the service request. There would thus be no pre- and no post-conditions for the *getTime* service of a clock.

3.1.1.1.8.3 An SLA for a caterer

We can package together the contract elements into a contract package. The package may be seen as a UML representation of a Services Level Agreement (SLA). It will contain the following elements:

- an interface specifying the services required from the service provider,
- potentially pre- and/or post-conditions on services,
- potentially quality requirements for the individual services or for the service provider as a whole (such quality requirements would apply across all services specified in the interface), and
- class diagrams for each of the objects exchanged between the client and the service provider.


Caterer contract (SLA)
Preconditions:
- order placed at least 3 hours prior to delivery.
- all invoices older than 7 days have been settled.
Caterer
+processLunchOrder(: Order) : Invoice
Postconditions:
- lunches delivered to delivery address.
- client account updated.
Order
-deliveryDateTime
-oderNumber
-deliveryAddress
Invoice
vendorVatNumber
invoiceDate
invoiceNumber
serviceDate
Quality requirements:
- scalability: must be able to cater for up to 40 people
- performance: lunches must be delivered within 10 minutes of the specified delivery time.
- modifiability: must be able to cater for various levels of vegetarians and for all major religious constraints.
- reliability: must maintain a failure rate of below 0.1%
LineItem
-itemIdentifier
-quantity
-unitPrice
*

A UML representation for a Service Level Agreement (SLA) for a caterer.

Figure 3.4: The Service Level Agreement (SLA) for a caterer

3.1.1.1.8.4 Always specify the contract from the perspective of the client

The purpose of the SLA is to encapsulate the client's requirements for a service provider, decoupling the client's business process from any particular service provider. The pre- and post-conditions and quality requirements should thus be specified from the perspective of the client's needs and not from the perspective of how a particular service provider is able to render a service.

3.1.2 Functional aspects

As per design-by-contract, the functional aspects of a service is described by the pre- and post-conditions, and the structure of the exchanged value objects. The core WSDL standard allows for the description of the functional aspects around a service.

3.1.2.1 WSDL (The Web Services Contract)

WSDL (the Web Services Decription Language) is an XML vocabulary to describe services contracts. It has become the key enabler of systems integration, by containing a full description of a service

- **abstractly,** by specifying the services interfaces, including the use-cases, pre-conditions, and structure of exchanged value objects, and
- **concretely,** by specifying how the abstract service is bound to a particular protocol (such as SOAP/HTTP), and where the service endpoints are.

3.1.2.1.1 The structure of a WSDL Document

A WSDL document consists of an `definitions` element in the WSDL namespace:

```
http://schemas.xmlsoap.org/wsdl/
```

The XML schema which specifies the structure of WSDL 1.1 documents (the most widely used version) can be found at:

```
http://schemas.xmlsoap.org/wsdl/wsdl.xsd
```

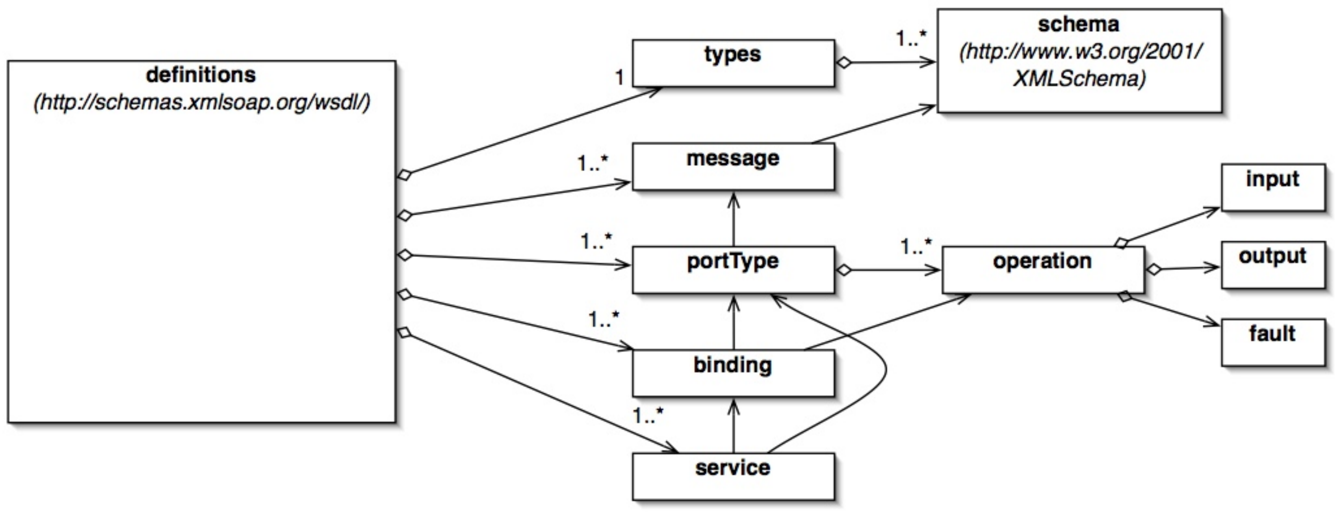


Figure 3.5: The structure of a WSDL Document

A WSDL *targets* a particular XML name space - which is, of course, XML's packaging mechanism: All of the declared elements in the WSDL can thus by uniquely referenced through a combination of their local name and name space.

A typical WSDL document contains up to five top-level sections:

1. **types** Contains (or imports from an external location) one or more schemas which specify the structure of exchanged value objects.

2. **messages** Declares all the exchanged value objects (request, response and fault messages) and binds them to a schema type or element reference.
3. **portType** The interface for the service. Indicates the operations (*use cases*) the service provides, and the messages exchanged for each.
4. **binding** Specifies how the abstract `portType` (and all the operations) is realised in a particular transport layer and messaging style.
5. **service** Indicates to clients where they can connect to start interacting with a particular binding of a particular portType, by means of a URL (in the case of HTTP).

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    targetNamespace="..."

    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/
                        http://schemas.xmlsoap.org/wsdl/wsdl.xsd">


    <!-- Define Types -->
    <wsdl:types>
        <xs:schema
            targetNamespace="..."
            elementFormDefault="qualified"
            attributeFormDefault="unqualified">
            ...
        </xs:schema>
    </wsdl:types>


    <!-- List exchanged messages -->
    <wsdl:message name="...">
        <wsdl:part name="body" element="..."/>
    </wsdl:message>
    ...


    <!-- Abstract Service Interface Definition -->
    <wsdl:portType name="...">
        <!-- Specify the use cases, and the messages exchanged -->
        <wsdl:operation name="...">
            <wsdl:input message="..."/>
            <wsdl:output message="..."/>
            <wsdl:fault name="..." message="..."/>
        </wsdl:operation>
        ...
    </wsdl:portType>


    <!-- Specify how our abstract service should be bound to a protocol
    such as SOAP/HTTP -->
    <wsdl:binding name="..." type="...">
        ...
    </wsdl:binding>


    <!-- Concrete service endpoints -->
    <wsdl:service name="...">
```

```
        <wsdl:port name="..." binding="...">
      ...
    </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

3.1.2.1.1.1 The ’types’ section

The `types` element encloses data type definitions that are relevant for the exchanged messages. WSDL uses W3C XML Schema to specify the types of exchanged messages, although - in principle - any other schema language could be used in the future.

```

<definitions .... >
    <types>
        <xsd:schema .... />*
    </types>
</definitions>

```

3.1.2.1.1.2 The ’message’ section

Messages consist of one or more logical parts. Each part is associated with a type from some type system using a message-typing attribute. The set of message-typing attributes is extensible. WSDL defines several such message-typing attributes for use with XML Schema:

- **element** Refers to an schema element using a *QName*. This is the WS-I recommended method for tying a message to its schema type, and it implies that the XML schema must contain element declarations for all high-level messages.
- **type** Refers to an schema `simpleType` or `complexType` using a *QName*. This is the conceptually better approach, but not WS-I recommended for interoperability reasons.

The syntax for defining a message is as follows. The *message-typing* attributes (which may vary depending on the type system used) are ‘element’ or ‘type’.

```
<definitions .... >
    <message name="nmtoken"> *
        <part name="nmtoken" element="qname"? type="qname"?/> *
    </message>
</definitions>
```

The message `name` attribute provides a unique name among all messages defined within the enclosing WSDL document.

The `part` name attribute provides a unique name among all the parts of the enclosing message. (usually not relevant, since Document/Literal SOAP messages only have one part.)

3.1.2.1.1.3 The ’portType’ section

A port type is a named set of *abstract operations* and the *abstract messages* involved.

```
<wsdl:definitions .... >
    <wsdl:portType name="nmtoken">
        <wsdl:operation name="nmtoken" .... /> *
    </wsdl:portType>
</wsdl:definitions>
```

The port type `name` attribute provides a unique name among all port types defined within in the enclosing WSDL document.

An operation is named via the `name` attribute.

WSDL has four *transmission primitives* (message exchange patterns) that an endpoint (web service) can support:

- **One-way** The endpoint receives a message.
- **Request-response** The endpoint receives a message, and sends a correlated message.
- **Solicit-response** The endpoint sends a message, and receives a correlated message.
- **Notification** The endpoint sends a message.

These are specified by altering the order and/or presence of the `input`, `output` and `fault` child elements.

3.1.2.1.1.4 The 'binding' section

A binding defines message format and protocol details for operations and messages defined by a particular `portType`. There may be any number of bindings for a given `portType`.

```
<wsdl:definitions .... >
    <wsdl:binding name="nmtoken" type="qname"> *
        <-- extensibility element (1) --> *
        <wsdl:operation name="nmtoken"> *
           <-- extensibility element (2) --> *
           <wsdl:input name="nmtoken"? > ?
               <-- extensibility element (3) -->
           </wsdl:input>
           <wsdl:output name="nmtoken"? > ?
               <-- extensibility element (4) --> *
           </wsdl:output>
           <wsdl:fault name="nmtoken"> *
               <-- extensibility element (5) --> *
           </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
</wsdl:definitions>
```

The `name` attribute provides a unique name among all bindings defined within in the enclosing WSDL document.

A binding references the `portType` that it binds using the `type` attribute.

Binding relies heavily on *extensibility elements* to provide binding-specific information, e.g. SOAP/HTTP.

3.1.2.1.1.5 SOAP/HTTP binding

The most common, and indeed the only safe (from an interoperability perspective) binding to use is the Document/Literal SOAP binding, which implies that SOAP messages are exchanged via HTTP. This binding is specified as:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  ...
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  ...

  <wsdl:binding name="MySOAPBinding" type="my:MyPortType">

    <!-- Indicate document-style messages over HTTP -->
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
```

```
        style="document"/>

    <!-- For each operation, indicate literal (XML Schema-based)
         encoding of the input, output and/or fault message(s) -->
    <wsdl:operation name="myOperation">
        <soap:operation soapAction="" style="document"/>
        <wsdl:input>
            <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output>
            <soap:body use="literal"/>
        </wsdl:output>
        <wsdl:fault name="nameOfFault">
            <soap:body use="literal"/>
        </wsdl:fault>
    </wsdl:operation>

  </wsdl:binding>

  ...

</wsdl:definitions>
```

3.1.2.1.1.6 The 'service' section

A service groups a set of related ports together:

```
<wsdl:definitions .... >
    <wsdl:service name="nmtoken"> *
        <wsdl:port .... />*
    </wsdl:service>
</wsdl:definitions>
```

The `name` attribute provides a unique name among all services defined within in the enclosing WSDL document.

Ports within a service have the following relationship:

- None of the ports communicate with each other (e.g. the output of one port is not the input of another).
- If a service has several ports that share a port type, but employ different bindings or addresses, the ports are alternatives. Each port provides semantically equivalent behavior (within the transport and message format limitations imposed by each binding). This allows a consumer of a WSDL document to choose particular port(s) to communicate with based on some criteria (protocol, distance, etc.).
- By examining it's ports, we can determine a service's port types. This allows a consumer of a WSDL document to determine if it wishes to communicate to a particular service based whether or not it supports several port types. This is useful if there is some implied relationship between the operations of the port types, and that the entire set of port types must be present in order to accomplish a particular task.

3.1.2.1.2 Importing resources into a WSDL

Since WSDL is just a container for any number and combination of *abstract* and *concrete* service description artifacts, it often makes sense to split these across two WSDL documents. This has the benefit of being able to publish *only one abstract WSDL document for the service contract*, which only needs to change when the service's functional requirements change.

On the other hand, the service's protocol bindings, endpoint locations, etc. continually evolve, and different service providers / consumers have their own WSDLs expressing these. They could all, however, refer to the same copy of the abstract service definition.

Another resource which is typically externally specified is the XML schema containing the type definitions. Though each service contract should be seen in isolation as supporting a specific use-case, it is common to re-use lower-level data objects across services contracts, and have the data objects specification maintained in a separate document.

3.1.2.1.2.1 Splitting WSDL abstract and concrete service descriptions

Based upon the functional requirements of a simple Clock, an abstract WSDL contract could be written as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://example.co.za/clock/"
  xmlns:clock="http://example.co.za/clock/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/
                      http://schemas.xmlsoap.org/wsdl/wsdl.xsd">

  <wsdl:documentation>
    An abstract contract for a clock
  </wsdl:documentation>


  <!-- Types used for exchanged value objects -->
  <wsdl:types>
    <xsd:schema targetNamespace="http://example.co.za/service/"
      elementFormDefault="qualified"
      attributeFormDefault="unqualified">

      <xsd:complexType name="GetTimeRequest"/>
      <xsd:element name="getTimeRequest" type="clock:GetTimeRequest"></xsd:element>

      <xsd:complexType name="GetTimeResponse">
        <xsd:sequence>
          <xsd:element name="time" type="xsd:time"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="getTimeResponse" type="clock:GetTimeResponse"/>

    </xsd:schema>
  </wsdl:types>


  <!-- Exchanged value objects and faults -->
  <wsdl:message name="GetTimeRequest">
    <wsdl:part  name="body" element="clock:getTimeRequest"/>
  </wsdl:message>
  <wsdl:message name="GetTimeResponse">
    <wsdl:part  name="body" element="clock:getTimeResponse"/>
  </wsdl:message>


  <!-- Service contract (interface) -->
  <wsdl:portType name="Clock">

    <wsdl:operation name="getCurrentTime">
      <wsdl:input message="clock:GetTimeRequest"/>
      <wsdl:output message="clock:GetTimeResponse"/>
    </wsdl:operation>

  </wsdl:portType>

</wsdl:definitions>
```

In the abstract contract, no mention is made to transport protocols or service endpoints, it is *purely functional*. In a separate WSDL document, we specify a particular binding and a particular service endpoint where clients may use a clock:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://example.co.za/clock/concrete/"
  xmlns:concrete="http://example.co.za/clock/concrete/"
  xmlns:clock="http://example.co.za/clock/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/
                      http://schemas.xmlsoap.org/wsdl/wsdl.xsd">

  <wsdl:documentation>
    A concrete contract for a clock, accessible via SOAP/HTTP
  </wsdl:documentation>


  <!-- Import all elements from the abstract contract -->
  <wsdl:import namespace="http://example.co.za/clock/" location="abstractContract.wsdl"/>


  <!-- Here we specify a SOAP/HTTP binding -->
  <wsdl:binding name="ClockSOAP" type="clock:Clock">

    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document"/>

    <wsdl:operation name="getCurrentTime">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>

  </wsdl:binding>

  <!-- Physical service endpoint(s) -->
  <wsdl:service name="ClockServiceSOAP">
    <wsdl:port name="Clock" binding="concrete:ClockSOAP">
      <soap:address location="http://www.clock.com/services/clock" />
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

3.1.2.1.2.2 Defining the structure of exchanged value objects in a separate schema

Though it is recommended that the request/response messages for the services be directly specified as part of the WSDL (for they are not re-usable outside of the service being described), the embedded schema may, of course, import any number of external schemas using the `<xsd:import ... />` mechanism.

This is commonly used for lower-level, re-usable data types which are shared between contracts.

3.1.2.1.3 Versions of WSDL

The current (widely-used) version of WSDL, version 1.1, was created through an industry co-operative effort lead by Microsoft and IBM. Ironically, this widely-used standard is not a W3C Recommendation, and remains a ‘W3C Note’.

WSDL version 2.0, which is a W3C Recommendation, is not widely supported yet (though it does explicitly form the basis of many SOA and Integration-related standards), although support for both versions can be expected for the foreseeable future. Version 2.0 introduces the following changes:

- Adds further semantics to the description language, such as interface specialisation
- Removes message constructs (an unnecessary source of clutter)
- Renamed a number of elements to more logical counterparts, such as `PortType` to `Interface`, and `Port` to `Endpoint`.

3.1.2.1.4 A simple example of a WSDL contract

As a simple example of a WSDL 1.1 contract, consider the following contract for a generic unit converter:

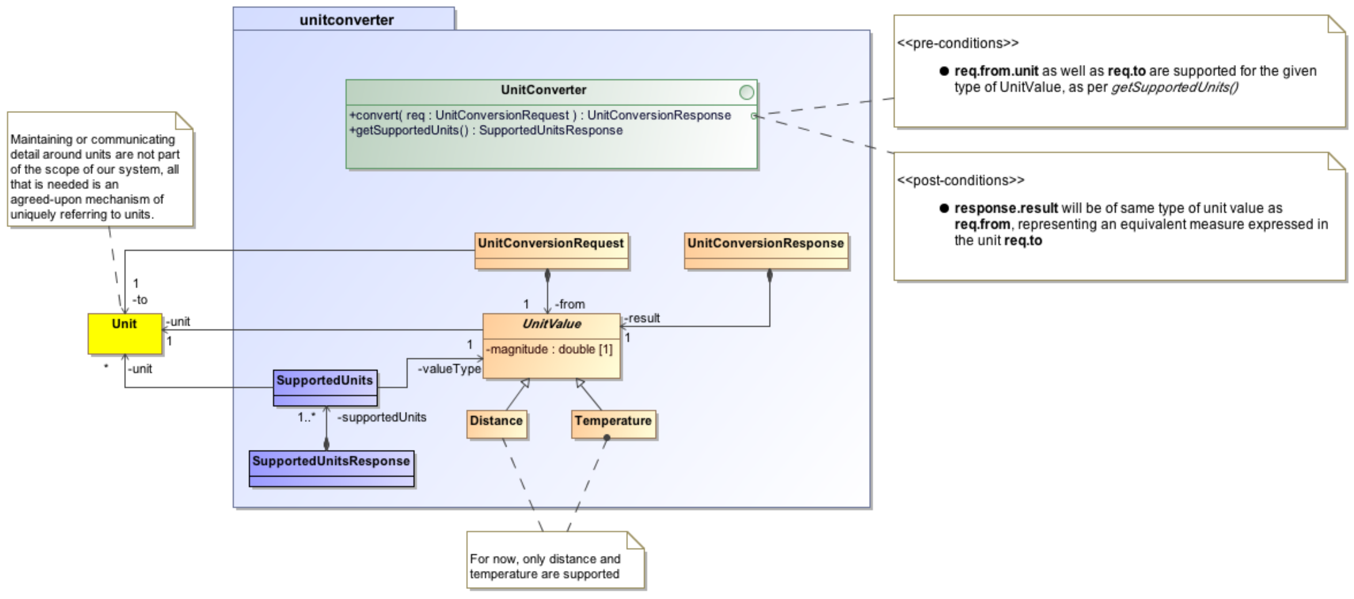

unitconverter
UnitConverter
+convert(req : UnitConversionRequest) : UnitConversionResponse
+getSupportedUnits() : SupportedUnitsResponse
<<pre-conditions>>
req.from.unit as well as req.to are supported for the given type of UnitValue, as per getSupportedUnits()
<<post-conditions>>
response.result will be of same type of unit value as req.from, representing an equivalent measure expressed in the unit req.to
Maintaining or communicating detail around units are not part of the scope of our system, all that is needed is an agreed-upon mechanism of uniquely referring to units.
UnitConversionRequest
UnitConversionResponse
1
-to
1
-from
-result
1
Unit
-unit
1
UnitValue
-magnitude : double [1]
1
-valueType
*
-unit
SupportedUnits
1..*
-supportedUnits
SupportedUnitsResponse
Distance
Temperature
For now, only distance and temperature are supported

Figure 3.6: A Contract for a generic unit converter

A `UnitValue` abstractly represents a measurement in some unit. It offers two services, one to query the units which are supported for each type of *unit value* (indicating which units it could be converted to) and another to perform the actual conversion.

Different types of unit value ar supported via specialisations of the abstract `UnitValue` class, with only two (distance and temperature) currently supported.

Note

Though the contract refers (via association) to units (such as kilometer, celcius) the information around units are beyond the scope of what we wish to build: Our types will only refer to types of units, and in our implementation mapping we need to decide on a mechanism to do so. In this case, we introduce a new simple type called `UnitReference`, which is a specialisation of string. URIs ar usually a good candidate for references to external concepts, in the spirit of RDF.

A WDL containing the abstract model of the service definition could look as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  name="unitconverter"
  targetNamespace="http://example.co.za/unitconverter/"
  xmlns:uc="http://example.co.za/unitconverter/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/
                      http://schemas.xmlsoap.org/wsdl/wsdl.xsd">

  <wsdl:documentation>
    This contract specifies the abstract service model for a simple
    but generic unit converter.
  </wsdl:documentation>


  <!-- Types used for exchanged value objects -->
  <wsdl:types>
    <xsd:schema>
      <xsd:import namespace="http://example.co.za/unitconverter/"
        schemaLocation="unitconverter.xsd"/>
    </xsd:schema>
  </wsdl:types>


  <!-- Bind Exchanged value objects and faults to types -->
  <wsdl:message name="UnitConversionRequest">
    <wsdl:part  name="body" element="uc:unitConversionRequest"/>
  </wsdl:message>
  <wsdl:message name="UnitConversionResponse">
    <wsdl:part  name="body" element="uc:unitConversionResponse"/>
  </wsdl:message>
  <wsdl:message name="UnsupportedUnit">
    <wsdl:part  name="body" element="uc:unsupportedUnit"/>
  </wsdl:message>
  <wsdl:message name="SupportedUnitsRequest">
    <wsdl:part  name="body" element="uc:supportedUnitsRequest"/>
  </wsdl:message>
  <wsdl:message name="SupportedUnitsResponse">
    <wsdl:part  name="body" element="uc:supportedUnitsResponse"/>
  </wsdl:message>


  <!-- Service contract (interface) -->
  <wsdl:portType name="UnitConverter">

    <wsdl:operation name="convert">
      <wsdl:input message="uc:UnitConversionRequest"/>
      <wsdl:output message="uc:UnitConversionResponse"/>
      <wsdl:fault name="unsupportedUnit" message="uc:UnsupportedUnit"/>
    </wsdl:operation>

    <wsdl:operation name="getSupportedUnits">
      <wsdl:input message="uc:SupportedUnitsRequest"/>
      <wsdl:output message="uc:SupportedUnitsResponse"/>
    </wsdl:operation>

  </wsdl:portType>

</wsdl:definitions>
```

The included schema which specifies the structure of the exchanged value objects:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
```

```
    targetNamespace="http://example.co.za/unitconverter/"
    xmlns:uc="http://example.co.za/unitconverter/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

    <xs:annotation>
        <xs:documentation>
            This schema specifies types for the exchanged value objects
            of a simple and generic unit conversion service.
        </xs:documentation>
    </xs:annotation>

    <xs:complexType name="UnitConversionRequest">
        <xs:sequence>
            <xs:element name="from" type="uc:UnitValue"/>
            <xs:element name="to" type="uc:UnitReference"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="unitConversionRequest" type="uc:UnitConversionRequest"/>

    <xs:complexType name="UnitConversionResponse">
        <xs:sequence>
            <xs:element name="result" type="uc:UnitValue"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="unitConversionResponse" type="uc:UnitConversionResponse"/>

    <xs:complexType name="UnsupportedUnit"/>
    <xs:element name="unsupportedUnit" type="uc:UnsupportedUnit"/>

    <xs:complexType name="SupportedUnitsRequest"/>
    <xs:element name="supportedUnitsRequest" type="uc:SupportedUnitsRequest"/>

    <xs:complexType name="SupportedUnitsResponse">
        <xs:sequence>
            <xs:element name="supportdUnits" type="uc:SupportedUnits"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="supportedUnitsResponse" type="uc:SupportedUnitsResponse"/>

    <xs:complexType name="UnitValue" abstract="true">
        <xs:sequence>
            <xs:element name="magnitude" type="xs:double"/>
        </xs:sequence>
        <xs:attribute name="unit" type="uc:UnitReference" use="required"/>
    </xs:complexType>

    <xs:simpleType name="UnitReference">
        <xs:restriction base="xs:string"/>
    </xs:simpleType>

    <xs:complexType name="Distance">
        <xs:complexContent>
            <xs:extension base="uc:UnitValue"/>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="Temperature">
        <xs:complexContent>
            <xs:extension base="uc:UnitValue"/>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="SupportedUnits">
        <xs:sequence>
            <xs:element name="valueType" type="uc:UnitValue"/>
            <xs:element name="unit" type="uc:UnitReference"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

</xs:schema>
```

Note

Any synchronous service is forced into the single-request/single-response paradigm. Even for a service such as `getSupportedUnits`, which does not accept any input arguments, one must introduce a request message in the WSDL/Schema.

Finally, a concrete WSDL, which specifies a SOAP/HTTP binding

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  name="unitconverter"
  targetNamespace="http://example.co.za/unitconverter/soap/"
  xmlns:uc="http://example.co.za/unitconverter/"
  xmlns:ucsoap="http://example.co.za/unitconverter/soap/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/
                      http://schemas.xmlsoap.org/wsdl/wsdl.xsd">

  <wsdl:documentation>
    This concrete service model specifies a SOAP binding and service endpoint for
    a unit converter (defined in a separate abstract contract).
  </wsdl:documentation>


  <!-- Refer to abstract service model -->
  <wsdl:import namespace="http://example.co.za/unitconverter/"
    location="unitconverter.wsdl"/>


  <!-- Specify a SOAP/HTTP binding -->
  <wsdl:binding name="UnitConverterSOAPBinding" type="uc:UnitConverter">
    <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>

    <wsdl:operation name="convert">
      <soap:operation soapAction=""/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
      <wsdl:fault name="unsupportedUnit">
        <soap:fault use="literal" name="unsupportedUnit"/>
      </wsdl:fault>
    </wsdl:operation>

    <wsdl:operation name="getSupportedUnits">
      <soap:operation soapAction=""/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>

  </wsdl:binding>


  <!-- Specify a concrete service endpoint, listening at a particular address -->
  <wsdl:service name="UnitConverterService">
    <wsdl:port name="UnitConverterSOAP"
      binding="ucsoap:UnitConverterSOAPBinding">
      <soap:address
        location="http://localhost:8080/utilities/unitconverter" />
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

3.1.2.1.5 Mapping a UML-based contract to WSDL

A services contract specified in UML can effectively be mapped to a WSDL *abstract service description* in a step-by-step manner.

3.1.2.1.5.1 Steps to be performed

1. Introduce an XML name space which corresponds to the package (owner) of the interface, and create a WSDL document which targets that namespace.
2. If any service has more than one *input* or *output* arguments, introduce a *wrapper* type which allows for the passing of the arguments as a single message.
3. Map each of the input and output arguments of each service to an XML schema type, which is embedded or linked to from the WSDL document.
4. For each precondition of each service, create an XML schema complex type which represents a *Fault*, which would be thrown if a service is refused because the precondition was not met.
5. For each request, response and fault which can be exchanged between the client and the service, introduce a corresponding WSDL message, which refers to the appropriate XML schema data type.
6. For the service provider interface, introduce a corresponding WSDL portType, containing WSDL operations for each service provided. Each service indicates the messages (referring to the messages above) that can be exchanged, as well as the calling semantics (synchronous vs. asynchronous). For each precondition, a fault is declared.

This covers the *functional* aspects of the WSDL document. All that remains is to decide on the *binding* (transport layer / protocol to be used), and most clients would likely want a SOAP (document/literal) binding, as prescribed by the WS-I basic profile.

3.1.3 Non-functional aspects

Since non-functional (quality) requirements can only be honoured by the infrastructure hosting the service, the service container will, in many cases, add policy assertions (using the `WS-Poicy` and `WS-PolicyAttachment` standards) to the published WSDL.

This, for example, is used to assert a service's transactional or security characteristics.

3.1.4 Exercises

1. Design a simple service contract for an `AddressBook`, which allows the client to
   - add a contact
   - remove a contact
   - update a contact
   - search for contacts by name

   A contact can either by a person or a company.
2. Implement your contract in WSDL 1.1 using a suitable WSDL editor, such as an XML editor. Separate the abstract contract from the concrete contract.

3.2 Implementing a service

Once a contract for a service exists, one needs to decide whether one wants to out-source the responsibility to an existing external service provider (in which case one may need to build an adaptor to it) or whether one wants to implement the service oneself.

3.2.1 Choosing an implementation technology

When building a service based on a contract, an implementation technology should be used which satisfies as many criteria of suitability as possible, such as that it

- *can realise the required qualities*, such as performance, scalability, reliability, security, maintainability
- *provides or has access to the required support technologies*, such as persistence, manipulation of graphics, multi-threading
- *is in a domain suitable to the nature of the service implementation,* such as being efficient at service orchestration, or at data transformation
- *in a technology with which the implementor(s) are proficient, or could acquire the skills in the required time-frame*

3.2.2 Implementation Technology Recommendations

Without taking into account the skills of the development team, the following recommendations can be made for a few typical service provider categories:

3.2.2.1 High-level workflow controller

If a service offers a high-level use-case (such as accepting an insurance claim) it may have to control a business processes which is realised by orchestrating many lower-level service providers. Furthermore, the business process may run for a potentially long time, with many interactions.

Assuming our requirements are

- easy maintainability
- ability to orchestrate many lower-level services
- ability to maintain complex workflow state
- high scalability
- low performance

a `WS-BPEL` process, deployed into a Business Process Execution Engine, may be found suitable to this task.

3.2.2.2 Lower-level service provider

If a service offers a service at a finer level of granularity (which may potentially be used in many contexts), such as performing a calculation, or storing / retrieving information from a database, our requirements may be:

- high performance
- high scalability
- access to support technologies, such as persistence
- no requirement to maintain business process state

A Java class may very well be suited to the task. To realise high scalability, plus easy access to support technologies such as persistence, an EJB is usually recommended for enterprise systems.

3.2.2.3 Transformation service

If a service primarily involves transforming one data structure to another, our requirements may be:

- high performance
- easy maintainability of transformation rules
- no need to access support technologies such as persistence
- no need to orchestrate lower-level service providers

An XSLT transformation document, deployed to a suitable service container which can associate a transformation service with a service contract, may be suited to the task.

3.2.2.4 Ingenuitive service

Where a contract has been sufficiently specified in terms of pre- and post-conditions, but the process of realising the service is too complex to define or model in a semantically meaningful way, and we are happy with:

- poor performance
- poor scalability
- poor reliability
- high adaptability
- high ingenuity

we may wish to out-source the service to one or more people, accessed via an adaptor which communicates with the person through some or other communication channel, such as e-mail, graphical user interface, etc.

3.3 Testing Services

If one follows a test-driven development methodology (an important component of agile software development), it is necessary to write one or more unit tests which tests our service.

3.3.1 What needs to be tested?

The unit test should ideally only test that the functional requirements of the service are satisfied, i.e. that

- when the preconditions to a service is met by a request,
- the postconditions must be realised.

The test should request the service in various ways to adequately cover the typically envisaged usage patterns.

3.3.2 What should not be tested?

When performing functional unit testing on a service, one does not want to test

- the lower-level infrastructure upon which it depends, such as object-relational persistence providers or e-mail services
- the infrastructure through which the service itself is requested, such as through an ESB,
- the protocol mapping / adaptor layer through which the service is requested, such as a web services framework / xml binding technology
- the lower-level services (if any) upon which this service depends.

Each of these *infrastructure* services will typically have their own tests (not implemented by the same developer who implemented the service), and one does not want to duplicate, for every business logic component that one writes, the testing of the infrastructure which it depends on.

The developer should trust that all lower-level services, as well as infrastructure services, are working as per their respective contracts. If we suspect otherwise, we should run the (separate) unit tests for those services.

3.3.3 Implementing the service test

When we follow the approach above of strictly separating the *functional* testing from the *non-functional* testing, we can implement the functional test in whichever testing framework is ‘closest’ to the service’s implementation technology. For example, if a service is implemented using the Java programming language, the JUnit testing framework would be very suitable.

In the future, with software being generated from a PIM (UML Platform-Independent Model) which has a semantically sound set of OCL constraints to express service pre- and post-conditions, we see tests being generated automatically for contracts.

In the context of SOA development, one may ultimately want to execute technology-neutral tests (against the WSDL contracts) which are then able to test one’s own service implementations, as well as out-sourced implementations, using the same test. Current web services testing frameworks do exist, but they are all tied to the concrete model of the WSDL, i.e. they make the assumption that a SOAP/HTTP transport protocol is used, making them less suitable both for other interaction protocols (File System, JMS, CORBA, ...). They all usually also inadvertently test the services infrastructure.

3.3.4 Integration testing

One of the principles of Services, in the context of SOA, is service autonomy. This means that, during the implementation or selection of a particular service provider, it needs to be tested in an isolated (non-integrated) manner. Only once services are assembled in a service assembly, ready for deployment to an integration infrastructure (such as an Enterprise Services Bus) should integration testing be performed.

Chapter 4

Enterprise Services Bus via JBI

4.1 The Java Business Integration (JBI) specification

4.1.1 Introduction

JBI is the result of an industry-wide effort to standardise business integration technologies and deliver them on the Java platform.

4.1.1.1 What is JBI?

JBI (package `javax.jbi`) is a public standard specification for an Enterprise Services Bus. It specifies a standard framework for hosting plug in components in a managed environment, and allows them to interact using a WSDL 2.0-based messaging system and the WSDL's services model. The goal is to allow users to obtain plug-in components that provide all the functions needed to provide an integration solution, without getting locked into a particular vendor. Components are portable across JBI implementations. Instead of the developer having to write all the integration functionality, one simply deploys configurations to the plug-in components.

JBI is often referred to as a 'container of containers', as many other containers (service engines, binding components) are plugged in, and to some extent managed by, a JBI environment.

JBI aims to be the standard for Java-based services oriented enterprise architectures in a similar way as Java EE has become the Java standard for object-oriented enterprise architectures.

4.1.1.2 Who supports / does not support JBI?

At this stage most of the role players within the Java-based SOA community are behind JBI with the notable exceptions of IBM and BEA, who are promoting their own SCA (Service Component Architecture) standard, which is quite different to JBI's more typical mediated message exchange ESB-based model.

4.1.2 JBI architecture

JBI (Java Business Integration) specifies a standard integration framework which can form the basis of a Java-based ESB (Enterprise Services Bus). The primary elements are

- **A normalised message router (NMR),** an infrastructure for message delivery and routing, which offers automatic coupling of services based on what each provides and/or requires from the environment.
- **Components,** containers which host deployed services in some or other technology, and for each deployment automatically publishes to the bus the services which is provided to the environment, as well as which services are required from the environment.

4.1.2.1 JBI Components

JBI components host services, and indicate to the ESB which services are provided, as well as which services are required (consumed), by the deployments made to the component. Each component may host services in a particular technology, such as via Java objects, BPEL business processes, XSLT transformation, or by simply acting as a proxy to a service external to the environment.

Each component implements standard services from the perspective of the ESB (for lifecycle management, accepting messages, and so on), and it communicates with the bus via a standard *delivery channel*, a bi-directional, contract-driven construct which guarantees that different components can be installed to different ESBs, as long as they all implement JBI.

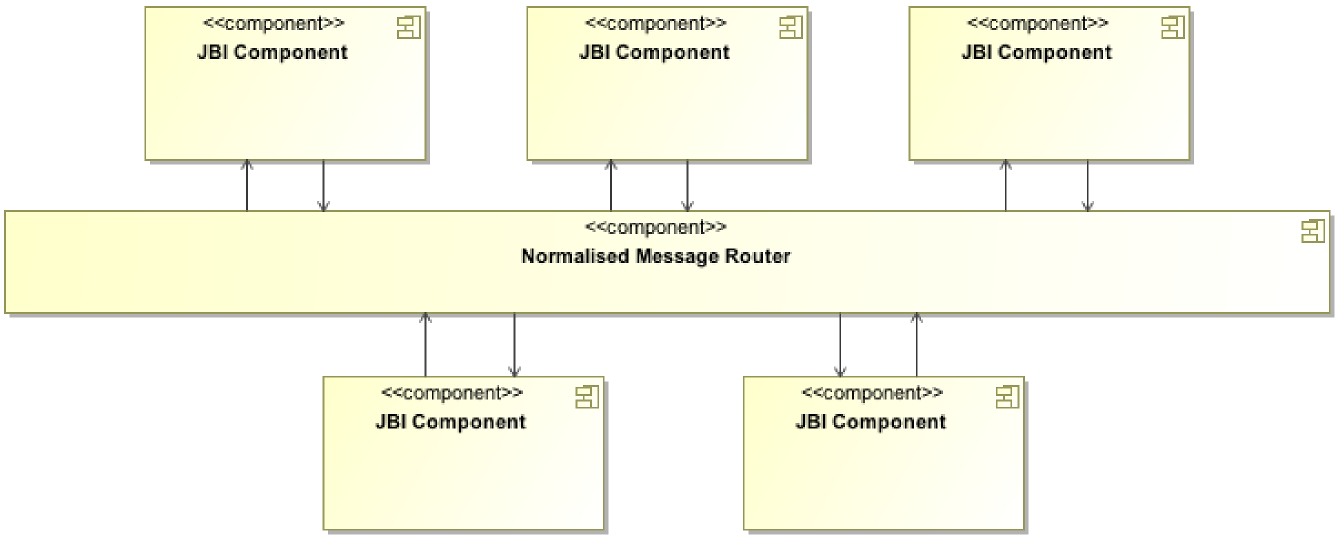


Figure 4.1: JBI components connected to the Normalised Message Router

4.1.2.1.1 Service Units

Component-specific artifacts, called *service units* (which are transparent to JBI) are deployed to each component, and the component is responsible to publish to the NMR what the services are that it *provides* (publishes to the infrastructure) and *consumes* (requires from the infrastructure), based on the contained service units:

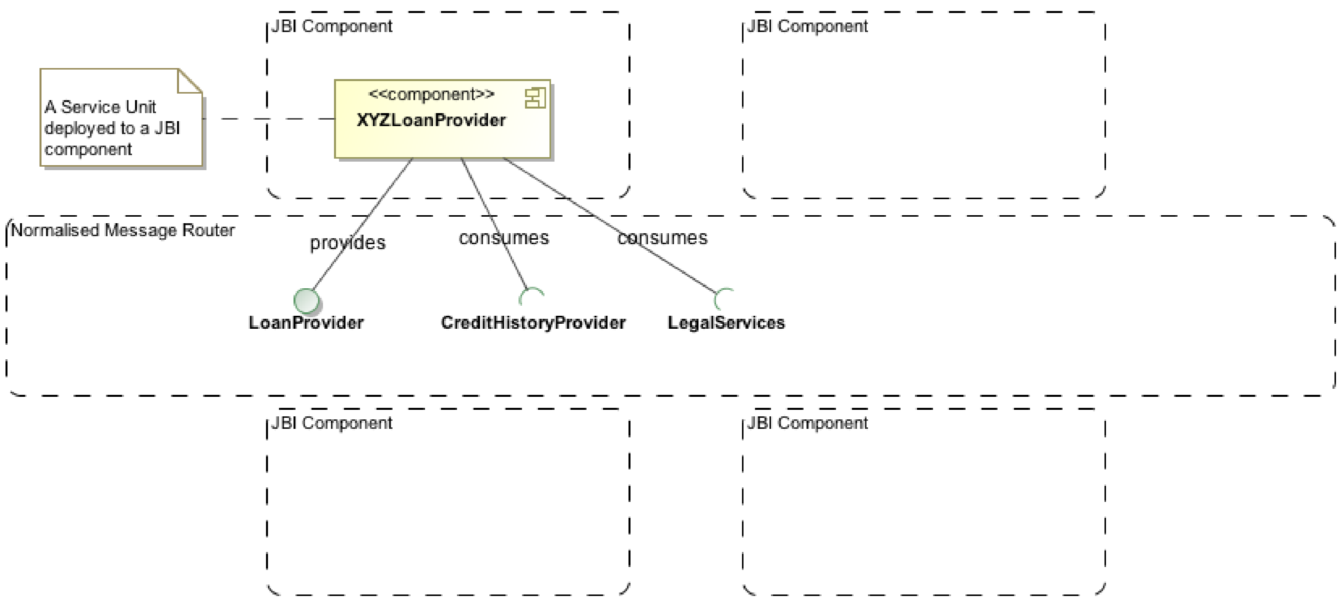


Figure 4.2: JBI Service units provide and consume services to/from the infrastructure

4.1.2.1.2 Types of JBI components

Though an arbitrary distinction from a technical point of view, JBI distinguishes between two different kinds of components,

- **service engines,** components which host service units that actually implement services / perform logic
- **binding components,** components which are responsible for transforming messages between an external protocol / format and the internal (normalised) messaging format, and which are used to
  - plug an external service provider in to the ESB, or
  - expose an endpoint so that clients external to the ESB can request services

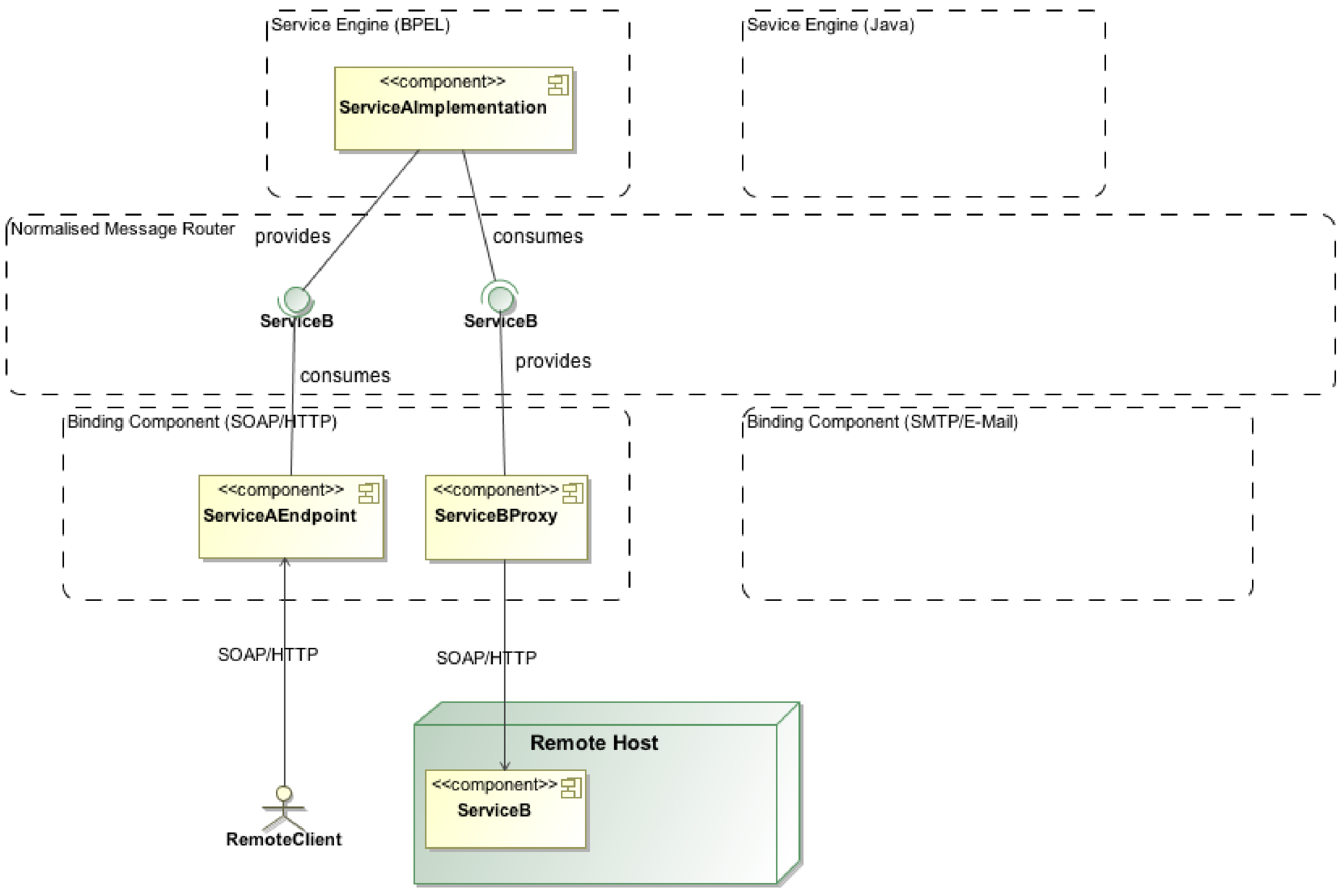


Figure 4.3: JBI Service Engines and Binding Components

4.1.2.2 Normalized messages

The normalized JBI message router routes normalized messages between plugin components. A normalized message contains two parts:

1. An abstract XML message complying to the abstract service model in WSDL.
2. Message meta data which allows processing elements to add (plug-in and system components) to add additional information to a message

4.1.2.2.1 Normalized message exchange

In JBI all messages are converted, via binding components, to normalized messages. The routing is done as per business process specification upon instruction from the services engine (for example a BPEL services engine).

The normalized message router supports message delivery with varying quality of service. This quality of service is typically set based on application needs and the nature of the message delivery channels. The normalized message router, supports the following quality of service levels:

- **Best effort** Messages may be dropped and they may be multiply delivered.
- **At least once** Each message is delivered at least once, but could be multiply delivered.
- **Once and only once** Each message is only delivered exactly once.

4.1.2.3 Management

The JBI environment including the components deployed within that environment (e.g. the services engines and binding components) are managed via JMX. This includes

- deployment of new components,
- deployment of component artifacts used by components,
- life cycle management (e.g. starting and stopping components),

and management and control of deployed components.

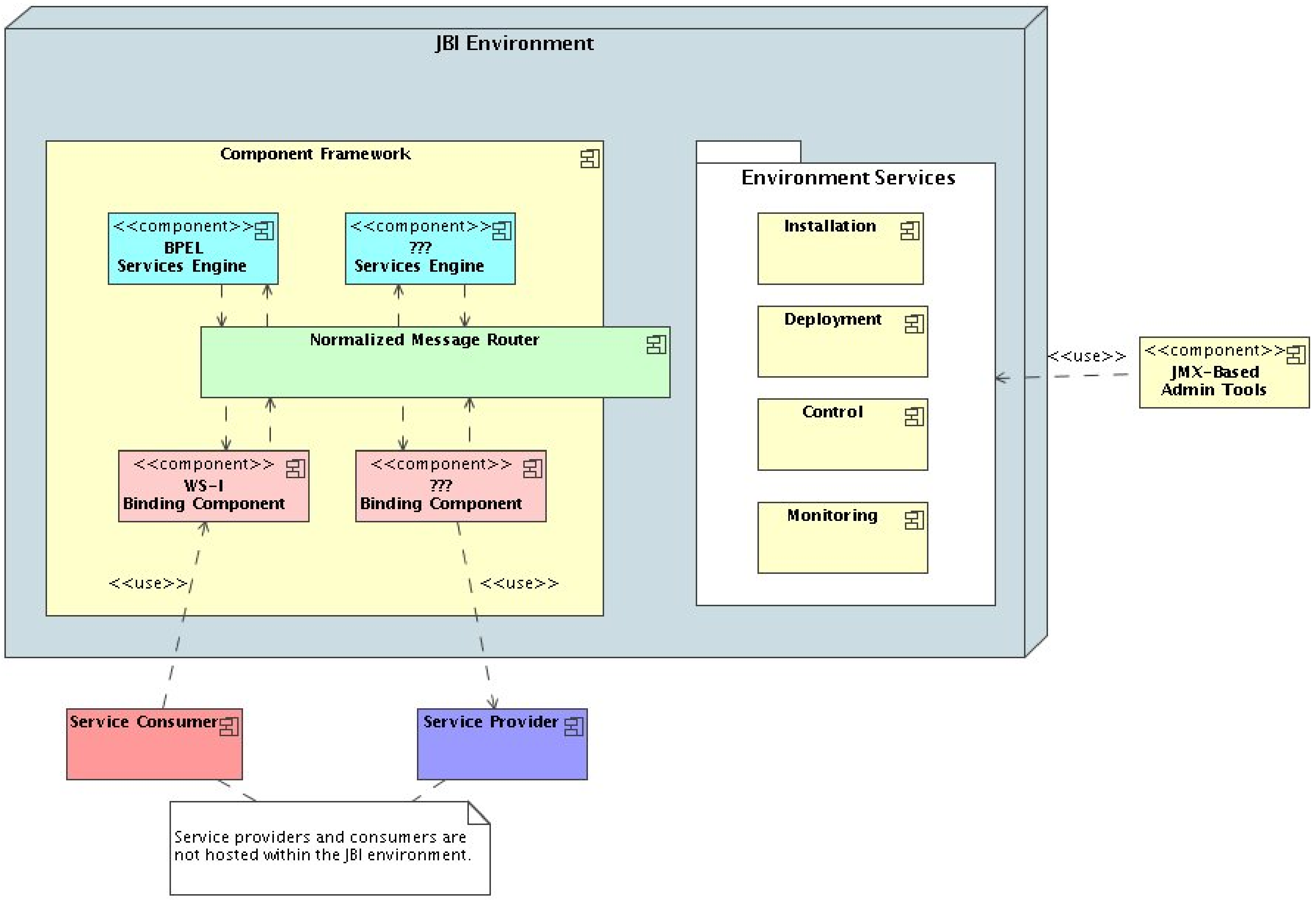

JBI Environment
Component Framework
<<component>> BPEL Services Engine
<<component>> ??? Services Engine
Normalized Message Router
<<component>> WS-I Binding Component
<<component>> ??? Binding Component
<<use>>
<<use>>
Environment Services
Installation
Deployment
Control
Monitoring
<<use>>
<<component>> JMX-Based Admin Tools
Service Consumer
Service Provider
Service providers and consumers are not hosted within the JBI environment.

Figure 4.4: JBI Management

4.1.3 The core elements of the JBI API

The JBI API provides a minimalist standardised framework for accessing the elements of a SOA in a standard way. It defines 4 packages and 28 interfaces. This section discusses the core elements defined by the JBI API.

Though this API is largely of interest to implementors of the specification (i.e. ESB vendors) certain service units (deployed in certain service engines) may use the JBI API directly to work with the ESB at a much lower level, such as exchanging messages, discovering service endpoints, and so on.

4.1.3.1 JBI components

A JBI component represents a service unit, i.e. a Java elements which provides a services to the bus. JBI defines a standard interface for `Components` which are meant to be deployed onto a JBI enabled ESB. This interface specifies standard service signatures for the following services:

- **getLifeCycle** Obtain a handle to the `ComponentLifeCycle` which enables one to `initialize`, `start`, `stop` and `shutdown` the component. In addition to this, the interface enables one to the name for the JMX bean for this component.
- **getServiceDescription** to retrieve the DOM object which contains the metadata description for the service provided by this component.
- **getServiceUnitManager** provides a handle to the object which manages this service unit's deployment.

4.1.3.2 NormalizedMessage

A `NormalizedMessage` provides a standard API for accessing (retrieving and setting) the elements of a message including

- the message content,
- the security context for a message,
- any message attachments, and
- any properties defined for the message.

4.1.3.3 MessageExchange patterns

Message exchange defines a standard interface for querying and executing a specific exchange of messages. It defines the names, sequences and cardinality of messages exchanged. JBI defines the following specialized message exchange patterns:

- **InOnly** a message exchange pattern for asynchronous service requests where there is only an input message an not an output message.
- **InOut** The message exchange pattern for a typical synchronous service request which blocks for a response message,
- **InOptionalOut** Message exchange patterns for to service providers which may optionally provide a response message.

A `MessageExchange` pattern provides the following services:

- query a unique identifier for this message exchange,
- retrieve the service address and operation name used by this exchange,
- query the message exchange pattern for this message exchange,
- queries whether the service requested through the message exchange falls within the context of a transaction manager,
- query the caller role for this exchange,
- a message exchange can be executed,
- query and set the status of the message exchange,
- create and set the fault for the message exchange and to query the fault message (if any) for the message exchange,
- get and set the normalized message to be used for a particular message identifier, and
- assign ans query any custom properties for the message exchange.

4.1.3.4 DeliveryChannel

The `DeliveryChannel` interface provides a standard API for dispatching a message exchange across some delivery channel.

4.1.4 Frameworks that implement JBI

JBI was finalised in August 2005, and a number of open-source and commercial implementations exist, such as

- Apache Servicemix
- OpenESB (part of the glassfish project)
- Petals ESB
- IONA
- Oracle

4.1.5 Service deployment and packaging

A typical JBI project might consist of several *service units*, all of which should be deployed / managed together as one integration solution. To this end, several service units are grouped together in a *service assembly*, a ZIP file together with some metadata which may be deployed / undeployed as a single unit.

4.1.5.1 Service units

A service unit contains artifacts which have meaning to the service engine for which it is destined, such as XSLT documents for a transformation engine, or Java classes for a Java container. The structure and/or contents are thus dictated by the service engine in question.

Each service unit must, however, contain a metadata file in `META-INF/jbi.xml` which indicates (by referring to service contracts, and/or specific service endpoint names) which services the unit *provides* and *consumes*. It is common for the `jbi.-xml` file to be generated by tools.

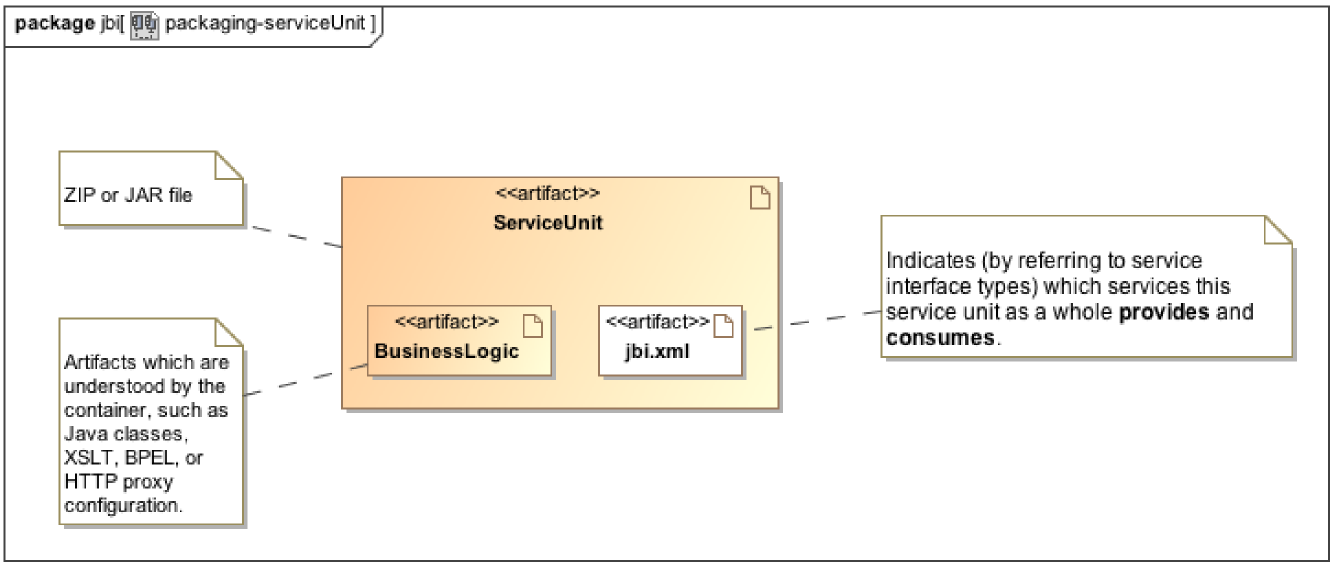

package jbi[packaging-serviceUnit]
ZIP or JAR file
<<artifact>>
ServiceUnit
<<artifact>>
BusinessLogic
<<artifact>>
jbi.xml
Artifacts which are understood by the container, such as Java classes, XSLT, BPEL, or HTTP proxy configuration.
Indicates (by referring to service interface types) which services this service unit as a whole provides and consumes.

Figure 4.5: Packaging a JBI Service Unit

An example of a `jbi.xml` file for a service unit is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <services xmlns:my="http://my.own/namespace/" binding-component="false">
    <provides interface-name="my:ServiceProviderA" service-name="my:ServiceProviderAService ↩
        " endpoint-name="ServiceProviderAEndpoint"/>
  </services>
</jbi>
```

Note
`service-name` and `endpoint-name` is as per a WSDL contract, and may be used to uniquely identify a service on the ESB

4.1.5.2 Service Assemblies

A service assembly contains an assembly of services, in the form of already-packaged *service units*:

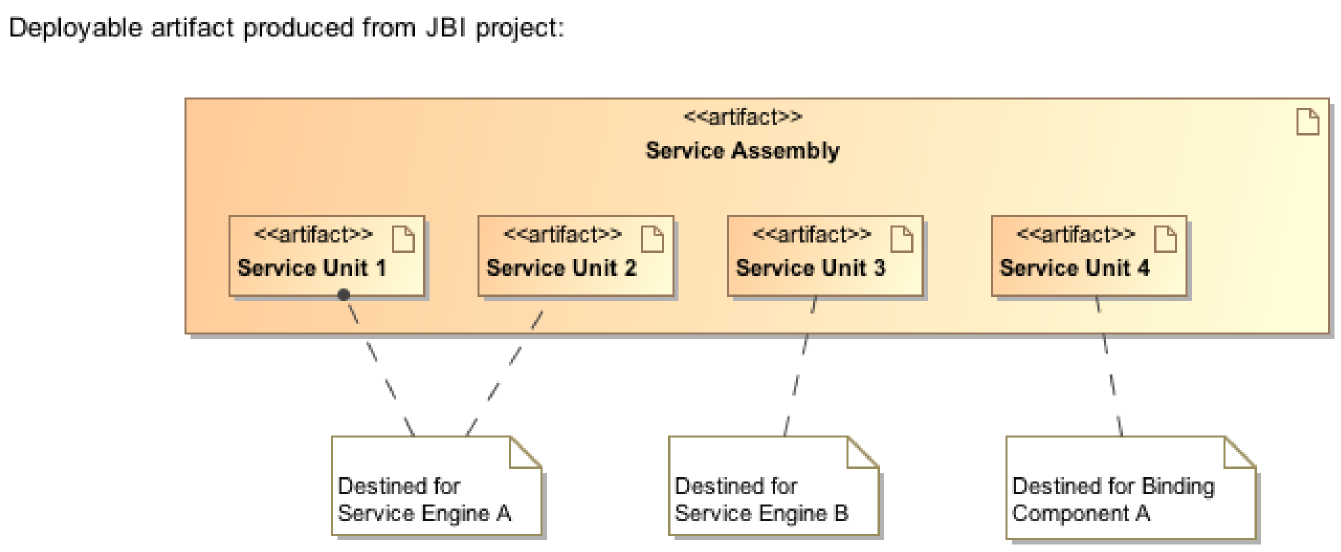

Deployable artifact produced from JBI project:
<<artifact>>
Service Assembly
<<artifact>>
Service Unit 1
<<artifact>>
Service Unit 2
<<artifact>>
Service Unit 3
<<artifact>>
Service Unit 4
Destined for Service Engine A
Destined for Service Engine B
Destined for Binding Component A

Figure 4.6: Packaging a JBI Service Assembly

A service assembly (itself a ZIP or JAR file) contains only one artifact in addition to the service units: A `jbi.xml` file which

- lists each service unit, as well as the name of the component to which it should be deployed, and
- optionally contains *service connections*, where the assembler has 'hard-wired' together specific service endpoints (as opposed to relying on the ESB's automatic message routing based on the service interface type)

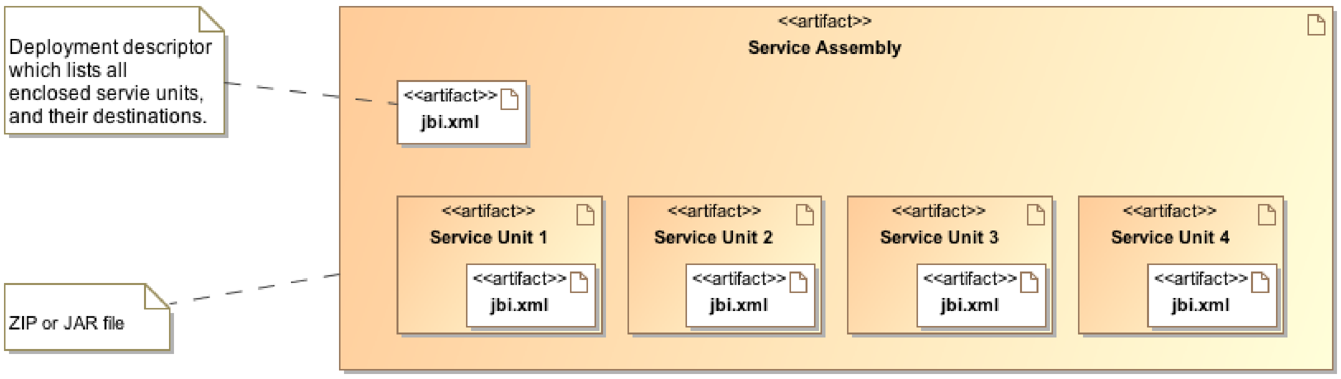


Figure 4.7: Packaging a JBI Service Assembly (Detail)

An example of a `jbi.xml` file (without service connections) for a service assembly is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <service-assembly>
    <identification>
      <name>my-integration-assembly</name>
      <description>My simple integration assembly</description>
    </identification>
    <service-unit>
      <identification>
        <name>serviceA-proxy</name>
        <description>A proxy for an external service provider</description>
      </identification>
      <target>
        <artifacts-zip>serviceA-proxy-1.0.zip</artifacts-zip>
        <component-name>servicemix-http</component-name>
      </target>
    </service-unit>
    <service-unit>
      <identification>
        <name>serviceB</name>
        <description>A BPEL-based implementation of ServiceB</description>
      </identification>
      <target>
        <artifacts-zip>serviceB.zip</artifacts-zip>
        <component-name>ode-bpel</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

4.1.5.3 Deployment

A service assembly can be deployed through one of a number of mechanisms, such as

- dropping the file in a *hot deploy* directory monitored by the JBI server,
- using the standard Apache Ant tasks to deploy over the network via JMX
- using Apache Maven to deploy over the network via JMX,

- using a ESB vendor-provided tool, usually in the form of plug-ins that integrate with your Java IDE

Each service unit will be deployed to its respective JBI component (binding component or service engine):

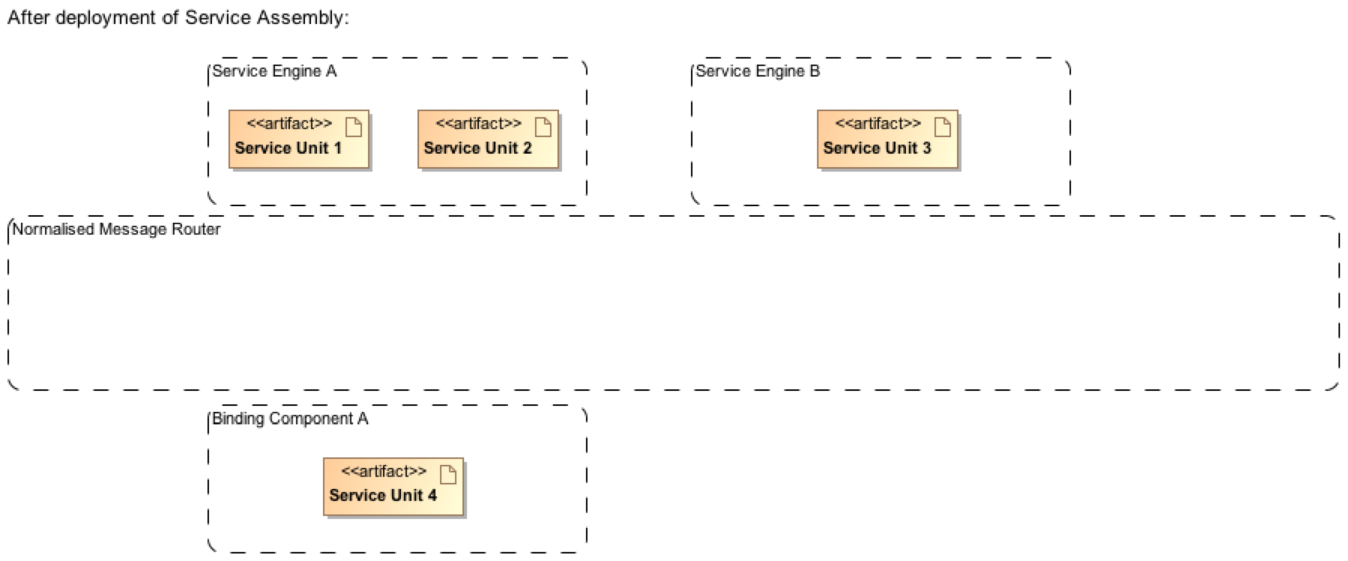

After deployment of Service Assembly:
Service Engine A
<<artifact>> Service Unit 1
<<artifact>> Service Unit 2
Service Engine B
<<artifact>> Service Unit 3
Normalised Message Router
Binding Component A
<<artifact>> Service Unit 4

Figure 4.8: JBI Deployment

4.1.5.3.1 Run-time endpoint activation

As each service unit is received by a JBI component, it must immediately perform *run-time endpoint activation* whereby the provided / consumed services are published to the ESB, which may then couple them together.

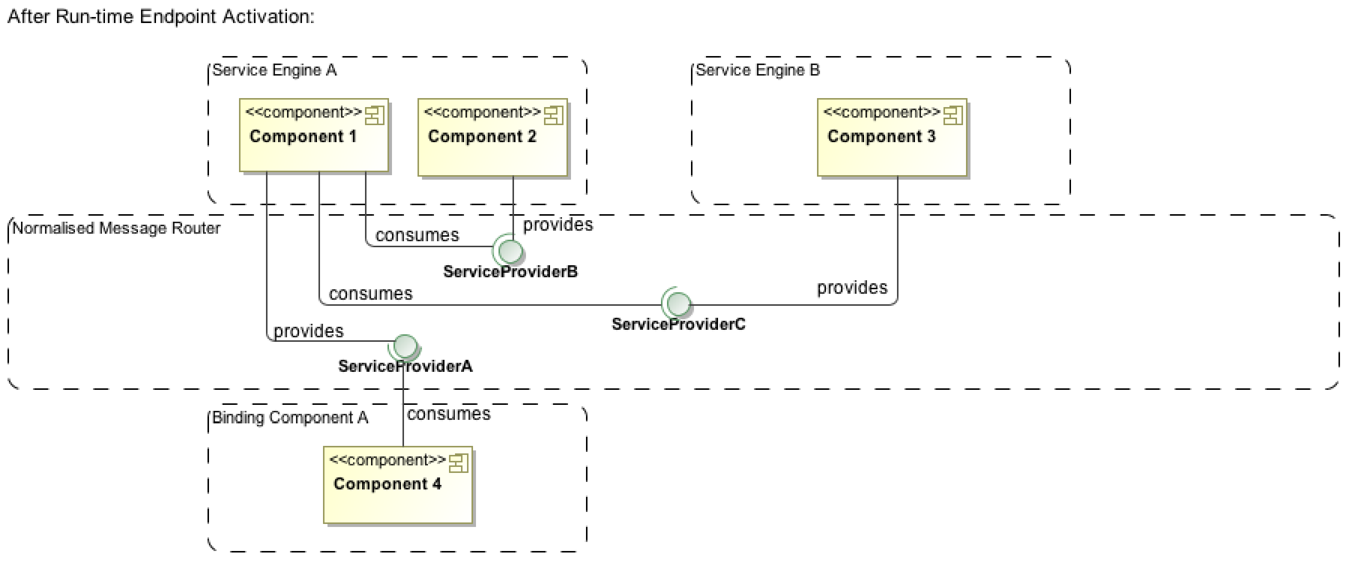

After Run-time Endpoint Activation:
Service Engine A
<<component>> Component 1
<<component>> Component 2
Service Engine B
<<component>> Component 3
Normalised Message Router
consumes
provides
ServiceProviderB
consumes
provides
ServiceProviderC
provides
ServiceProviderA
consumes
Binding Component A
<<component>> Component 4

Figure 4.9: JBI Run-time endpoint activation

4.1.6 What can JBI be used for ?

As its name (‘Java Business Integration’) implies, JBI lends itself to a wide range of integration scenarios. We here describe a number of common use-cases which any JBI-compliant ESB enable:

4.1.6.1 Eliminating point-to-point integration

Even though modern contract-driven software development de-couples clients from service implementations in terms of functionality, when it comes to certain scenarios - such as several of our internal systems using an external web service - we still face the risk of incurring a great deal of cost when, for example, the external services changes address, or (worse yet) when it ceases to exist, and we have to find an alternative (or build our own).

Instead of locking our internal systems to the one external address, we may instead

- deploy a proxy service unit (in a binding component such as SOAP/HTTP) that plugs the external service provider into our ESB, making it available to our internal systems. It thus *provides* a service, as per the provider’s service contract.
- deploy a service unit that exposes a web service endpoint at a stable, known internal address. This service unit *consumes* a service that implements the service provider’s contract.

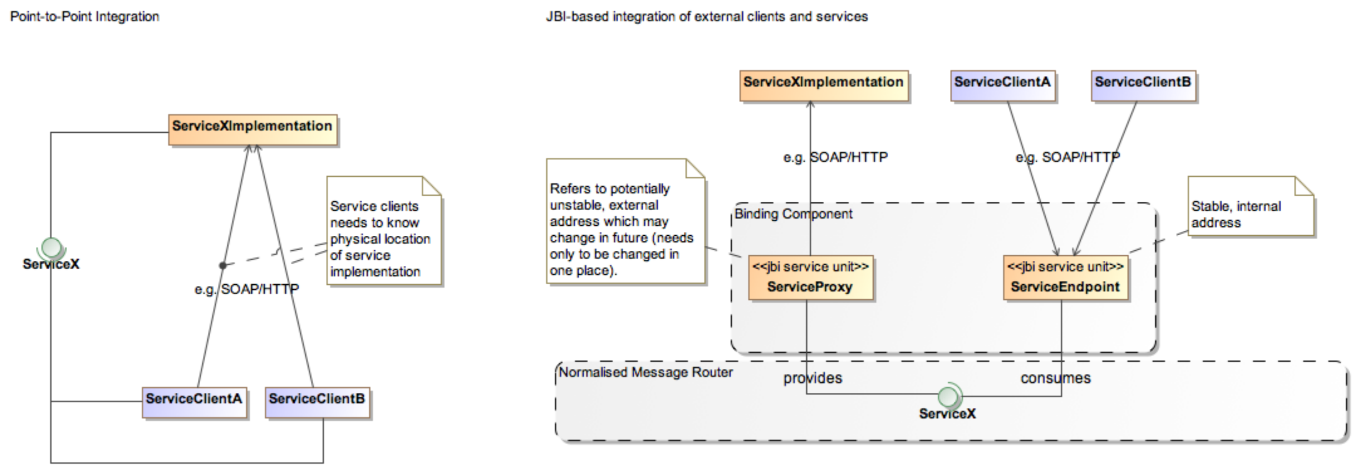

Point-to-Point Integration
ServiceXImplementation
ServiceX
Service clients needs to know physical location of service implementation
e.g. SOAP/HTTP
ServiceClientA
ServiceClientB
JBI-based integration of external clients and services
ServiceXImplementation
ServiceClientA
ServiceClientB
e.g. SOAP/HTTP
e.g. SOAP/HTTP
Refers to potentially unstable, external address which may change in future (needs only to be changed in one place).
Binding Component
<<jbi service unit>> ServiceProxy
<<jbi service unit>> ServiceEndpoint
Stable, internal address
Normalised Message Router
provides
consumes
ServiceX

Figure 4.10: Eliminating point-to-point integration

Should the external service change or disappear (forcing us to use another) we are guaranteed to only have to change the configuration of a single JBI service unit, referring to the new address. This scenario is particularly effective when the service clients themselves are not deployed in service engines plugged in to the ESB (such as web-based or stand-alone application clients, or even legacy systems).

Routing all requests through a single channel has the additional side benefit of being easily monitorable, or applying compression (to use less bandwidth), performing message validation, or gathering statistics. These may easily be added in the future, without the service, nor its clients, being directly aware.

4.1.6.2 Building Low-level Services

Instead of simply connecting external services to our ESB infrastructure, we can build implementations of services contracts and deploy them via the ESB to their respective service engines.

This in many ways sets JBI apart from other pure integration infrastructures (such as Apache Camel), and makes it very convenient to centrally manage and deploy large assemblies, which contain both business logic components, as well as integration constructs (protocol and functional adaptors, etc). By ‘low-level services’, we refer to atomic services which perform some useful, isolated function, such as storing or retrieving information in a database, performing calculations, etc. These are services which do not request other services via the ESB.

- It is essential that a WSDL contract describing the desired service first exists: Unlike certain technologies (like EJB) which can generate a WSDL contract from one’s programming code for convenience, it is not recommended to attempt to follow the same approach in JBI. Since so many clients may build dependencies on the service contract, it is best not to treat it as a temporary, generated artifact, but rather as the primary artifact from which we can, in return, generate programming code.
- Once a suitable implementation technology is chosen (preferably one that has access to all the necessary support technologies, such as database persistence, etc) the service is written, tested, and then deployed as a service unit that *provides* the service as per the service contract. In many cases, the service implementation’s programming language artifacts can be generated from the WSDL contract.

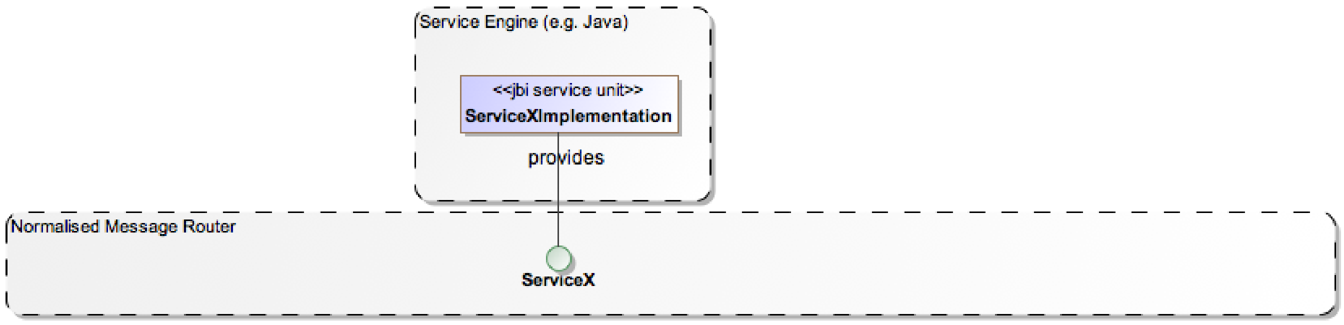


Figure 4.11: Building Low-level Services

4.1.6.2.1 Choosing an implementation technology

Since the service engine is fully responsible for mapping requests and responses from the Normalized message format to service calls in the respective programming language / technology, the developer is free to choose any implementation technology which is well-suited to the Job. Options available typically include

- Java (either plain Java objects, Spring Beans, or EJBs). Certain Java EE application server implementations (such as Glassfish) is particularly well-integrated with the ESB (e.g. OpenESB) to enable seamless deployment of all kinds of Java EE components via the ESB. EJB is typically a sensible choice because of the scalability and simple concurrency mae possible by the object and thread pooling.
- Scripting Languages (JavaScript/ECMAScript, Python, Groovy). Some of these languages offer simpler syntax, and simple yet powerful data manipulation / calculation facilities - but typically much weaker support technologies - compared to a mature general-purpose programming language like Java.
- XSLT. Certain services can be performed by performing very simple self-contained calculations or transformations on the input message, or by returning information stored in plain XML files. In such cases, XSLT offers a simple and high-performance alternative.

4.1.6.3 Requesting services via different protocols

Most services service their clients via a single protocol, such as SOAP/HTTP. We may wish to, however, enable such a service (which may be an external service, beyond our control) to be used by clients of various types, via different protocols.

Because a large number of JBI Binding components (each targeting a specific protocol) are available, and because they each convert messages to the NMR’s common normalised format, it is quite simple to expose a service via different protocols by

- Deploying a service unit that *provides* a certain service to the environment. This could either be an external service (by deploying a proxy to the service into the appropriate binding component, e.g. SOAP/HTTP) or a component that actually provides the service, such as a Java class to a Java container (recall that both cases are treated identically by the JBI environment)
- Deploying any number of message consumer endpoints in the binding components of your choice (E-Mail, JMS, CORBA, File/FTP, etc). Each of these service units *consume* the original service, and will automatically convert incoming messages to the normalised message format before dispatching them to the service.

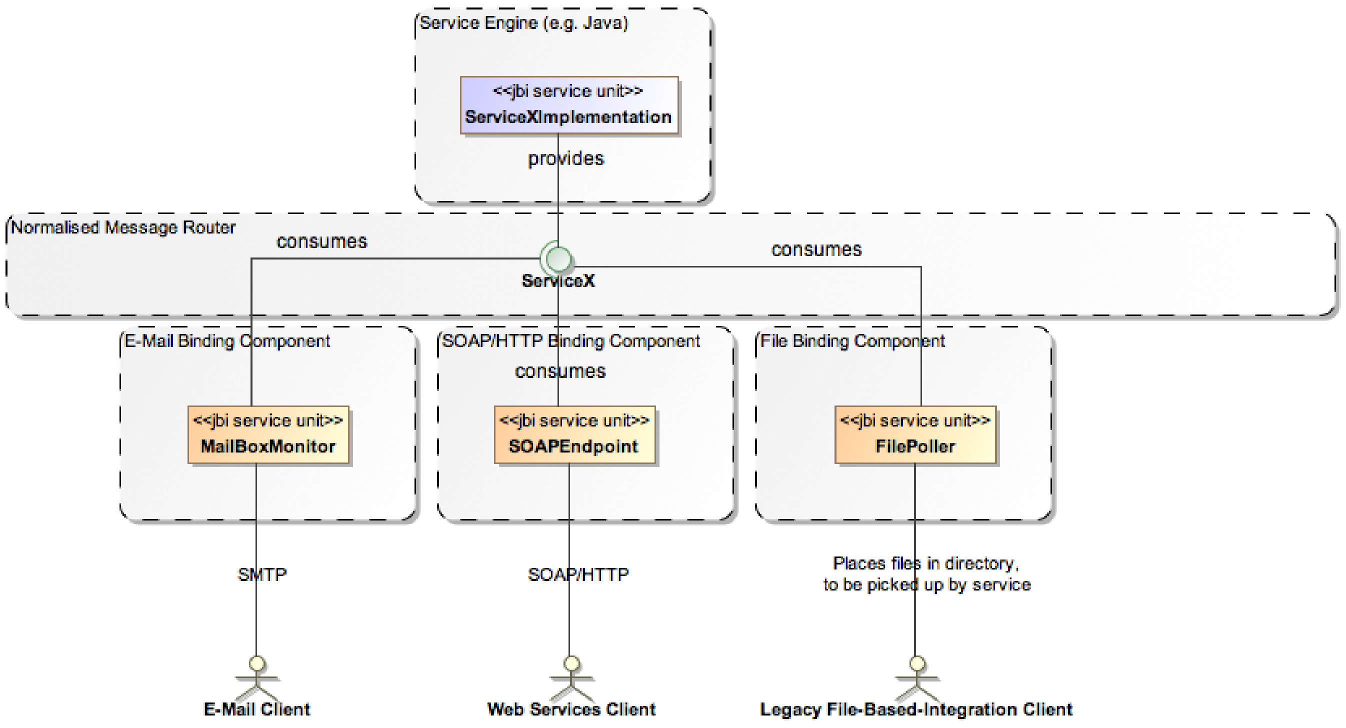


Figure 4.12: Requesting services via different protocols

Each binding component will specify its own allowable configuration to customise the nature of the exposed endpoint. For example, specifying the mailbox to monitor, the text-to-XML transformation rules, or the frequency with which a directory must be polled for new files.

4.1.6.4 Building Requirements-Driven Adaptors to incompatible services

A fundamental premise of contract-driven software development is that the contract *is specified from the perspective of the client*. That is to say, the contract *formally represents the client's requirements for a service*.

It is, after all, the client who wishes to ensure that he is never locked into using a particular service provider (the service provider would typically be very happy to have lifetime-guaranteed clients regardless of the quality of service it offers!). To this end, when we design the client, we first generate contract(s) for all the services required by the client's business processes. At this stage, we do not even know whether implementations that satisfy these contracts exist - we may indeed have to build implementations ourselves. When we do look for external services, we may find services that represent a close match - a particular provider may offer either a superset of the required functionality, or the service interface / exchanged data objects may be similar, but identical.

It is very important to, in such cases, *not directly couple the client to a particular service provider's contract.* If he did (and even if the match was extremely close or exact), the client would have to acknowledge

- that the provider's contract *exactly matches* the client's requirements,

- that the provider happens to change their contract (typically because the *provider* had undergone certain requirements changes) the client is happy to absorb those changes as being part of his/her requirements - immediately, unconditionally, and regardless of what those changes may entail. This will typically imply changes to the core business process of the client. And lastly,
- that if the provider ceases to exist, or happens to incorporate such radical changes that a change of provider is unavoidable, the client is happy to incur the cost of making core changes to his business processes to accommodate the new service provider.

Of course, this cycle could repeat itself very often. It is clear that most clients should not be built in such a manner. JBI offers us an environment in which any differences between what a client wants, and what a provider offers, can be absorbed in an adaptor. This localises any changes that are required to, for example, accommodate a new / different service provider. It also means that we can re-use our tests for any given service provider, because such tests have been written against the client's requirements, and not a particular provider's offerings. The following process could be followed:

- A client (either a service running in a service engine, or a consumer endpoint to accommodate clients external to the ESB) expresses a need to *consume* a certain service, based on a contract the client itself supplies.
- If there is currently no service that *provides* such a contract, we may look around and find, for example, an external service that offers a very similar kind of service. From a logical perspective, it may be exactly the service that we require, but the details of the service contract (or 'API') may be quite different. We may, for example, deploy a proxy to a binding component, indicating to the environment that it *provides* this service according to its contract.
- We may now go ahead an build an adaptor service unit which implements (*provides*) the client's contract, and which *consumes* the service actually provided by the service provider.

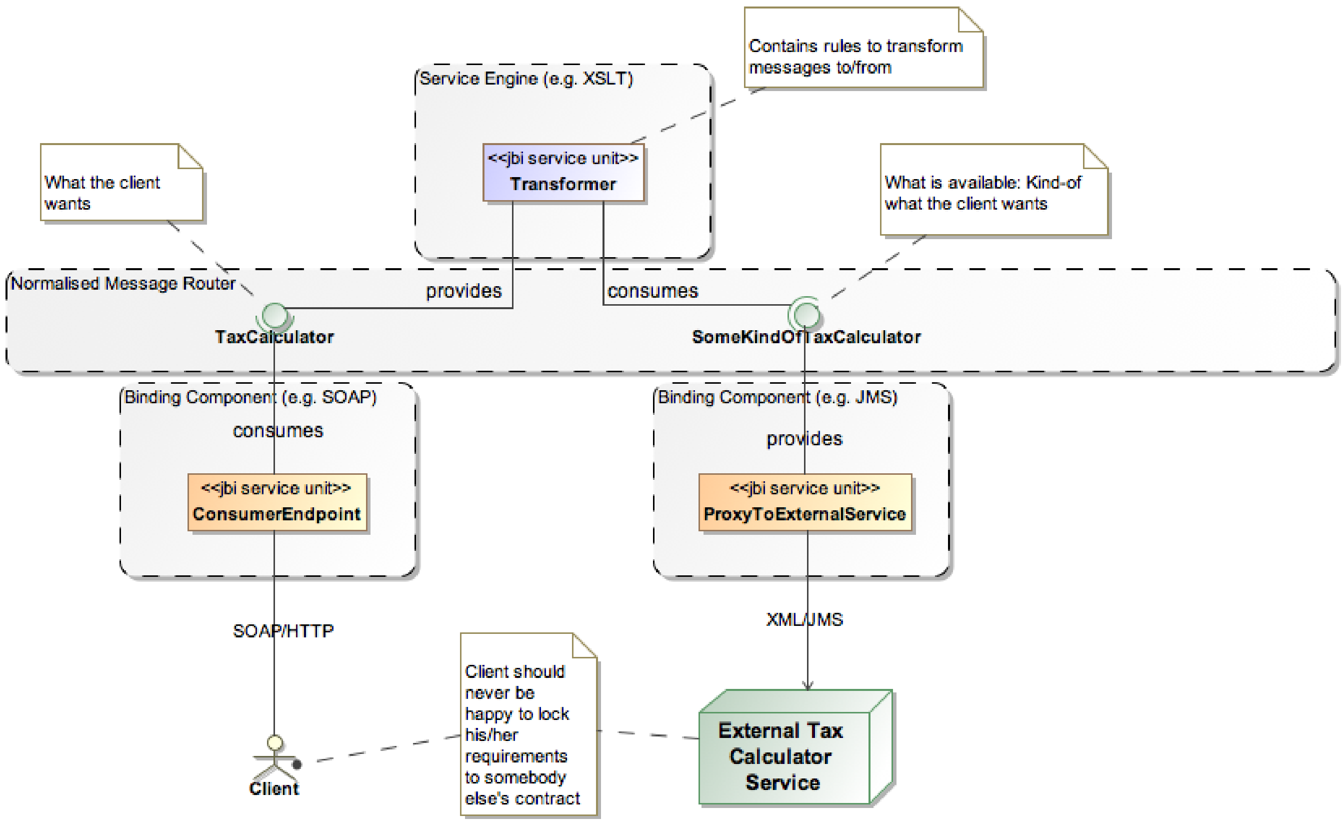

Contains rules to transform messages to/from
Service Engine (e.g. XSLT)
<<jbi service unit>>
Transformer
What the client wants
What is available: Kind-of what the client wants
Normalised Message Router
provides
consumes
TaxCalculator
SomeKindOfTaxCalculator
Binding Component (e.g. SOAP)
consumes
<<jbi service unit>>
ConsumerEndpoint
Binding Component (e.g. JMS)
provides
<<jbi service unit>>
ProxyToExternalService
SOAP/HTTP
XML/JMS
Client should never be happy to lock his/her requirements to somebody else's contract
Client
External Tax Calculator Service

Figure 4.13: Building Requirements-Driven Adaptors to incompatible services

4.1.6.4.1 Choosing an implementation technology

The adaptor may be built in any of a wide range of technologies, based on the required kind of adaptation. In its simplest form, an XSLT may specify mapping rules to adapt request and response messages (in a fully self-contained manner) between the two contracts. In its most complex form, an adaptor may require several other services, and/or support technologies such as database persistence - in which case it may be built in an orchestration language such as WS-BPEL, or even Java.

4.1.6.4.2 The Cost/Benefits ratio

It may seem like a large amount of unnecessary work and complexity up-front to, for every client, introduce its own contract(s) that encapsulate its requirements, and then build adaptors to adapt the client's needs to an actual service. This approach, however, is the only reasonable one to, in the long term, accommodate large numbers of client needs for 'the same' service (without forcing the same, complex API upon all clients), to ensure cost-effective maintainability of the clients' business processes, and manage risk around locking to particular services / vendors. In short, it is well worth the effort to keep things clean up-front.

4.1.6.5 Orchestrating Services

Service Orchestration is a blanket term for any service which itself makes use of (usually multiple) other services via the ESB. As such, it is not all that much of a special case, and it likely more common that any other type of service.

When viewed in the context of, say, the URDAD design methodology, we see that orchestration is usually performed by the *use-case controller* at a particular level of granularity. A use-case controller assembles a business process across multiple other services purely based on their contracts. The actual services may be hosted by a service engine on the ESB, or by an external party. Furthermore, the service (as invoked by the use-case controller) may in fact be offered by an adaptor that adapts its requirements to the offering of a particular provider.

JBI offers immense value in fully isolating a high-level service from the details of the services which it uses, in terms of

- location (internal or external to the environment)
- implementation technology
- transport protocol (if the services are external), and
- functional differences in terms of service contract, which may be absorbed by an adaptor that implement's the contract required by the high-level service

In practice, there are different types of high-level (orchestrator) services based on the implementation technology, such as

- short-running, often transactional services, similar to what one would build in straightforward Java / EJB, and
- long-running, stateful business processes which may span many client-service interactions.

Recall, however, that JBI itself treats the details of service units as 'black boxes', and in both cases the JBI environment only cares about the static service linkages, i.e. the provides / consumes relationships. So, regardless of implementation technology or style, a high-level service (such as an instance claim submissions service) may look as follows:

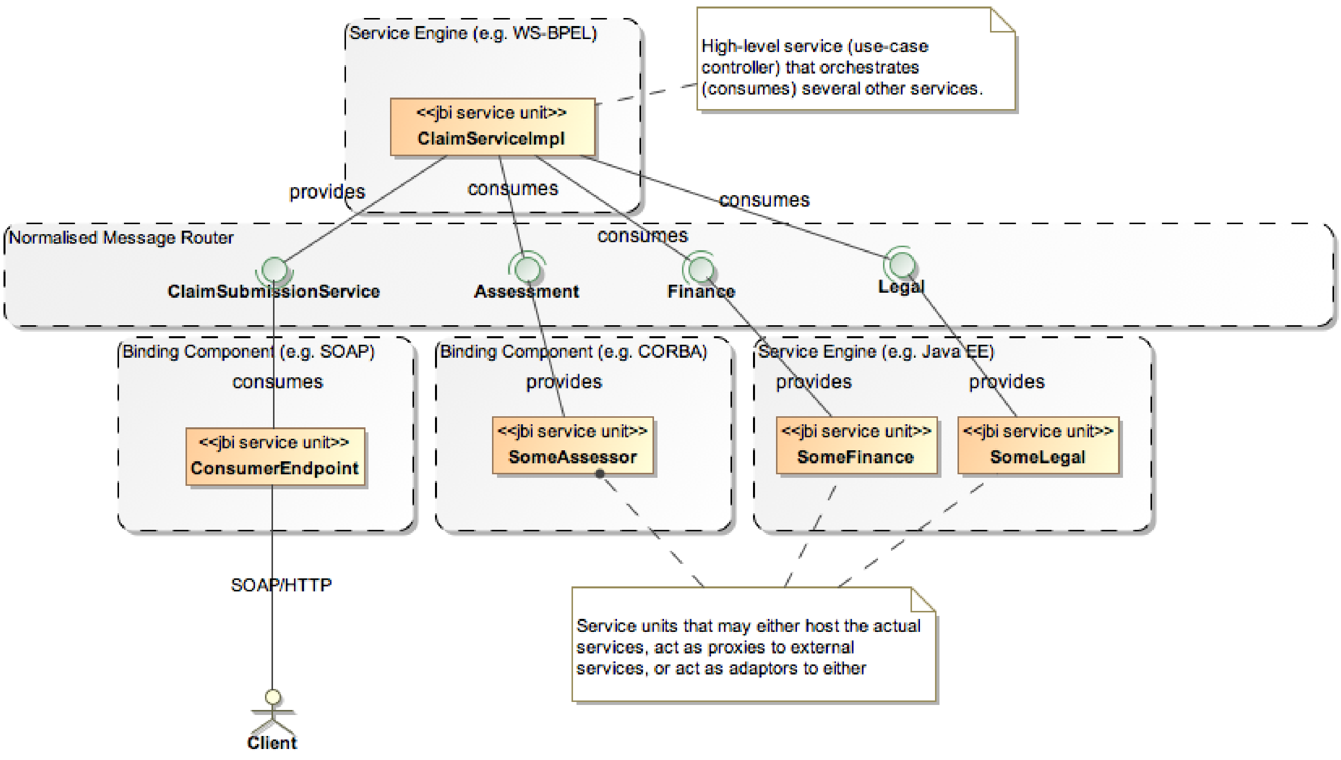

Service Engine (e.g. WS-BPEL)
<<jbi service unit>>
ClaimServiceImpl
High-level service (use-case controller) that orchestrates (consumes) several other services.
provides
consumes
consumes
consumes
Normalised Message Router
ClaimSubmissionService
Assessment
Finance
Legal
Binding Component (e.g. SOAP)
consumes
<<jbi service unit>>
ConsumerEndpoint
Binding Component (e.g. CORBA)
provides
<<jbi service unit>>
SomeAssessor
Service Engine (e.g. Java EE)
provides
provides
<<jbi service unit>>
SomeFinance
<<jbi service unit>>
SomeLegal
SOAP/HTTP
Service units that may either host the actual services, act as proxies to external services, or act as adaptors to either
Client

Figure 4.14: Orchestrating services

4.1.6.5.1 What defines orchestration ?

Since orchestration simply involves the invocation of other services via the ESB (which a number of other scenarios - such as building adaptors - also involves) one is tempted to ask why certain services are classified as performing orchestration, whilst other are not? Orchestration generally describes services implemented in a technology which can invoke other services with the following qualities amongst others:

- the high-level service can construct request messages to low-level services using any information at its disposal, including the output(s) of low-level services which were previously invoked
- the high-level service has mechanisms to deal with individual low level services failing or being refused (because a precondition was not met), and can itself follow a different business process to ‘deal’ with such failures (such as the `try { ... } c-atch (...) { ... }` block in the Java programming language.
- the high-level service has a mechanism(s) at its disposal to ensure the reliability and consistency of the business process, by ether enlisting low-level services in a transaction which can eb rolled back if necessary, or by being able to invoke compensating services to compensate for / undo partial failure (such as the `<compensationHandler>` element of WS-BPEL

4.1.6.5.2 Dynamic endpoint resolution

By default, most JBI orchestration services are analogous to EJB or Spring-based services, in that they declare a set of static service requirements to the environment (via each service unit’s `jbi.xml`), for which the environment ‘injects’ suitable service endpoints based on what is currently available. Each service thus specifies the services it would like to use up-front.

Some high-level services, however, do not know up-front which specific service providers they will use. A simple example could be a high-level home selling service, which would like to find and use all currently available real-estate agent services on the ESB. They will require the ability to look up service endpoints (by service name or type) on-the-fly, and most service engines / technologies that allow for orchestration provide a means for this, such as

- being able to assign instances of `WS-Addressing` endpoint references (which may have been created in the context of the business process) to service invocations, as in `WS-BPEL`, or
- providing programmatic access to the JBI API (such as in Java-based service providers) in order for the process to query the environment (similar to a Java JNDI lookup) for service endpoints.

4.1.6.6 Content-based routing

Content-based routing can be seen as a special form of Service Orchestration, where the router service

- receives a message, and is responsible for routing the message to one or more other services based on the contents of the message (or perhaps some other context, such as the day of the week, time of day, etc)
- does not have to control or be aware of any business process (and thus, does not concern itself with partial service failure, request message construction, etc), and thus
- does not necessarily implement a services contract

Although any orchestration technology could be used to implement a content-based router (because most of them allow for dynamic endpoint resolution) a content-based router can usually be implement in a service engine that specifically allows for the declarative specification of routing rules (e.g. in an XML file), such as the Apache Camel service engine. This usually allows the developer to specify message filers using the XPath query language, and route to the appropriate ultimate service provider.

If one wants benefit from JBI's automatic service coupling based on services contracts, it may make sense to introduce a services contract for the content-based-router which

- has a unique name
- specifies the exact services and exchanges messages as the ultimate services to which it routes messages

We may now plug multiple implementations of some service contract into our ESB. A service that requires them, does not *consume* their services contract, however. It *consumes* the content-based-router's contract, which will ensure that messages are routed to the ultimate service(s) based on the routing rules contained within the router service unit (e.g. a Apache Camel pipeline).

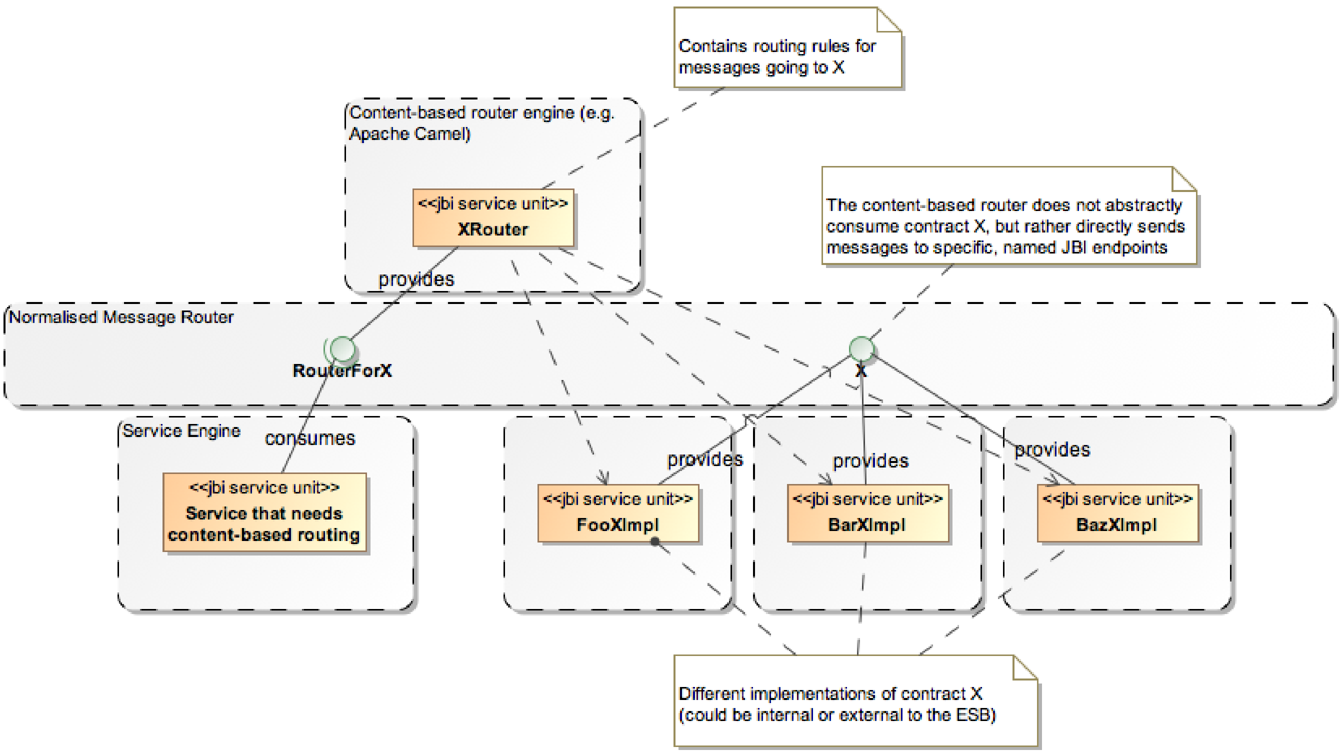

Contains routing rules for messages going to X
Content-based router engine (e.g. Apache Camel)
<<jbi service unit>> XRouter
provides
The content-based router does not abstractly consume contract X, but rather directly sends messages to specific, named JBI endpoints
Normalised Message Router
RouterForX
X
Service Engine
consumes
<<jbi service unit>> Service that needs content-based routing
provides
<<jbi service unit>> FooXImpl
provides
<<jbi service unit>> BarXImpl
provides
<<jbi service unit>> BazXImpl
Different implementations of contract X (could be internal or external to the ESB)

Figure 4.15: Content-based routing

4.1.6.7 Exercises

1. Consider two current integration use-cases that you are faced with in your organisation. Explain how JBI can realise each of these, by referring to the abstract JBI scenarios discussed in this section, and diagrammatically illustrate your proposed solutions.

4.1.7 Building a JBI Project with Apache Maven

Although JBI explicitly specifies standard targets for Apache Ant regarding the management and deployment of JBI Service Assemblies, there are a number of compelling reasons to use the more complex Apache Maven tool to manage a JBI project (provided the developer desires the 'independent' approach of not relying on all-encompassing graphical tools to manage one's JBI project):

- JBI project modules typically contain multiple sub-modules (service units and service assemblies) with potentially complex dependencies on one another as well as on libraries and frameworks which are used to build specific types of service units
- Instead of duplicating service information in several places (such as in *jbi.xml* files across service units and assemblies, as well as in the artifacts in the service units themselves, such as service endpoint configurations, we wish to have certain files generated for us
- As the specification as well as the implementations are still quite new, we would like tools which continually update themselves to evolve together with the JBI components which they support
- Maven's dependency management, and large catalogue of plugins, can help us with laborious tasks such as managing our WSDL contracts across service units.

4.1.7.1 The Apache Servicemix Tooling

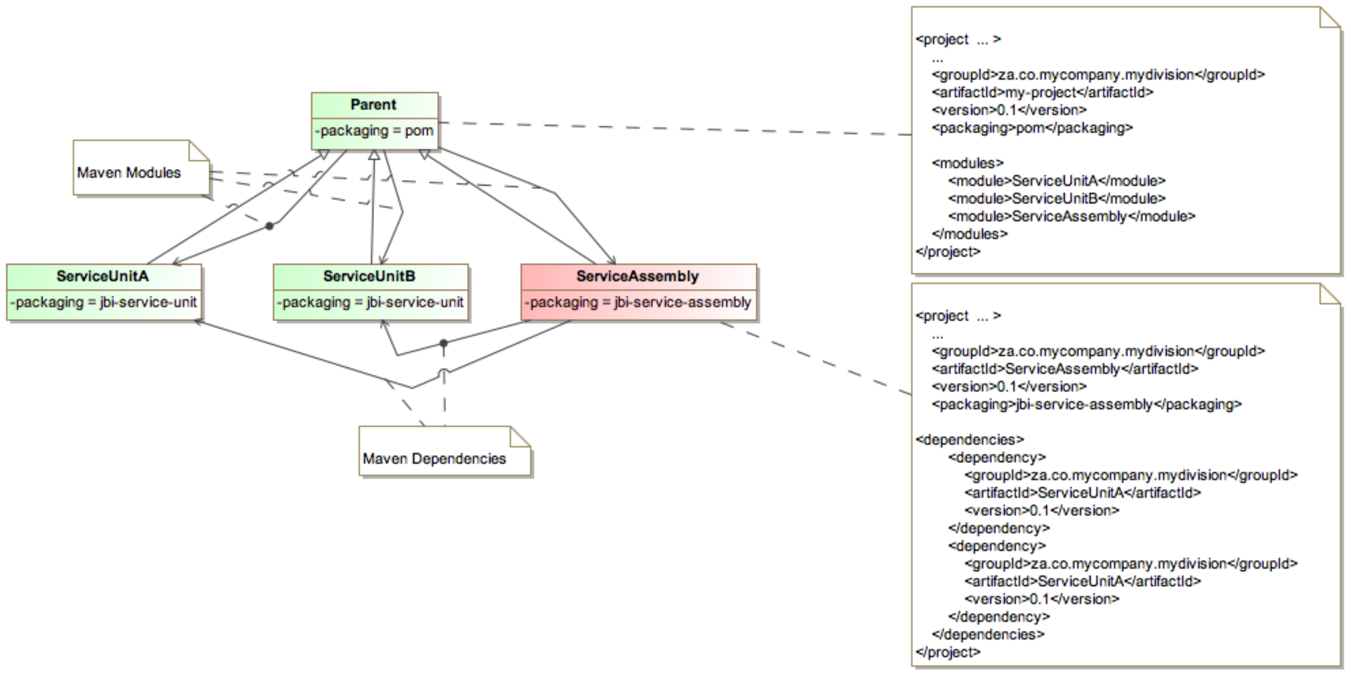
As part of the Apache ServiceMix ESB project, a set of JBI plugins have been developed for Maven (group Id `org.apache-.servicemix.tooling`) which support the development of JBI service units and service assemblies, taking over several of the mundane responsibilities such as generating the `jbi.xml` file(s) for us.

4.1.7.1.1 What is provided?

- Two new project packaging types, `jbi-service-unit` and `jbi-service-assembly`
- A set of plug-ins what are automatically bound to the correct Maven lifecycle phases, performing tasks such as generating the `jbi.xml` file for us after analysing the project's dependencies on other services, and correctly packaging the service units for deployment.

4.1.7.1.2 Project Structure

Maven allows for considerable freedom in terms of structuring the modules of one's projects. Typically, however, most Maven-based JBI projects of reasonable complexity consist of a parent project (POM), which contains multiple JBI service unit child projects, and at least one Service assembly with dependencies on the service units to include.



```
<project ... >
  ...
  <groupId>za.co.mycompany.mydivision</groupId>
  <artifactId>my-project</artifactId>
  <version>0.1</version>
  <packaging>pom</packaging>

  <modules>
    <module>ServiceUnitA</module>
    <module>ServiceUnitB</module>
    <module>ServiceAssembly</module>
  </modules>
</project>
```

```
<project ... >
  ...
  <groupId>za.co.mycompany.mydivision</groupId>
  <artifactId>ServiceAssembly</artifactId>
  <version>0.1</version>
  <packaging>jbi-service-assembly</packaging>

<dependencies>
    <dependency>
      <groupId>za.co.mycompany.mydivision</groupId>
      <artifactId>ServiceUnitA</artifactId>
      <version>0.1</version>
    </dependency>
    <dependency>
      <groupId>za.co.mycompany.mydivision</groupId>
      <artifactId>ServiceUnitA</artifactId>
      <version>0.1</version>
    </dependency>
  </dependencies>
</project>
```

Figure 4.16: Maven JBI Project Structure

4.1.7.1.3 Creating a new parent project

A new empty directory is created, containing a default POM with project metadata as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>za.co.mycompany.mydivision</groupId>
  <artifactId>my-project</artifactId>
  <version>0.1</version>
  <packaging>pom</packaging>

  <name>My Big Project (TM)</name>
  <url>http://mycompany.co.za/myproject</url>
</project>
```

4.1.7.1.4 Creating service units

In the parent project directory, we can created different kinds of *service units* based on the Maven archetype (project template) we specify. For example:

```
mvn archetype:create -DarchetypeGroupId=org.apache.servicemix.tooling -DarchetypeArtifactId ↩
    =servicemix-service-unit -DartifactId=my-service-unit
```

Some valid values for the `archetypeArtifactId` property are

- **servicemix-service-assembly** A service-assmebly module that will contain all other service units listed as dependencies
- **servicemix-cxf-bc-service-unit** A service unit that exposes a SOAP/HTTP endpoint for external clients and forwards the messages to a service hosted on the ESB, or a service unit that plugs an externally available SOAP/HTTP service into the ESB as a provided service.
- **servicemix-cxf-se-service-unit** A service implemented in Java (POJO/Bean)
- **servicemix-ode-service-unit** A BPEL-based process for deployment into the ODE Business Process Execution Engine
- **servicemix-saxon-xslt-service-unit** XSLT transformation-based service
- **servicemix-camel-service-unit** A service unit that will perform content-based routing using Apache Camel
- **servicemix-mail-service-unit** An endpoint which accepts incoming message via e-mail, or plugs an external e-mail based service into the ESB.

4.1.7.1.5 Adding service units to a service assembly

A Service Assembly must explicitly specify which service units are to be packaged are part of it. This is to facilitate explicit control over the decomposition of one's integration solution into several easily-managed service assemblies, instead of forcing everything into one large service assembly (which must always be deployed/un-deployed as a whole).

When the JBI Maven Plugin builds a project with packaging type `jbi-service-assembly`, it will package together all projects with packaging type `jbi-service-unit`. which have been specified as Maven dependencies in the POM:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>za.co.mycompany.mydivision</groupId>
      <artifactId>my-service-unit</artifactId>
      <version>0.1</version>
    </dependency>
  </dependencies>
  ...
</project>
```

4.1.7.1.6 Building the project

Running

```
mvn install
```

on the parent project will build and assemble all modules, as well as install them in the local Maven repository. Note that, despite the curious naming, this is not the same as *deployment* to the ESB, it is merely making the built modules available to other modules in the Maven project(s).

4.1.7.1.7 Deploying the service assembly

After packaging, the service assembly can either by deployed by copying the ZIP file to the ESB's hot deployment directory, or it can be deployed via JMX using Maven, by running (in the service assembly module directory)

```
mvn jbi:projectDeploy -DforceUpdate=true
```

Note
At time of writing, repeated Maven-based deployment and undeployment is often found to be unreliable, leaving the ESB in a potentially inconsistent state. File-system based deployment to a 'hot deploy' directory is often a more reliable alternative.

4.1.8 Examples

The following examples illustrate some basic integration use-cases. They are all built using the Maven tooling for JBI, with service units targeting the service engines and binding components shipped with Apache Servicemix. These containers themselves could be installed on another JBI-compliant ESB, or the individual service units' configuration could be modified to allow deployment into the 'equivalent' components provided by other ESB implementations.

4.1.8.1 Publishing an external service to the ESB

Consider the scenario where an implementation of a `UnitConverter` service exists at some external location on the internet. We wish to connect that service to our services infrastructure, in order for our own components to be able to make use of such a service without them each having to perform point-to-point integration to the service. Furthermore, if we later discover a better implementation of the service, or decide to implement our own, we can change service providers without having to update any client services.

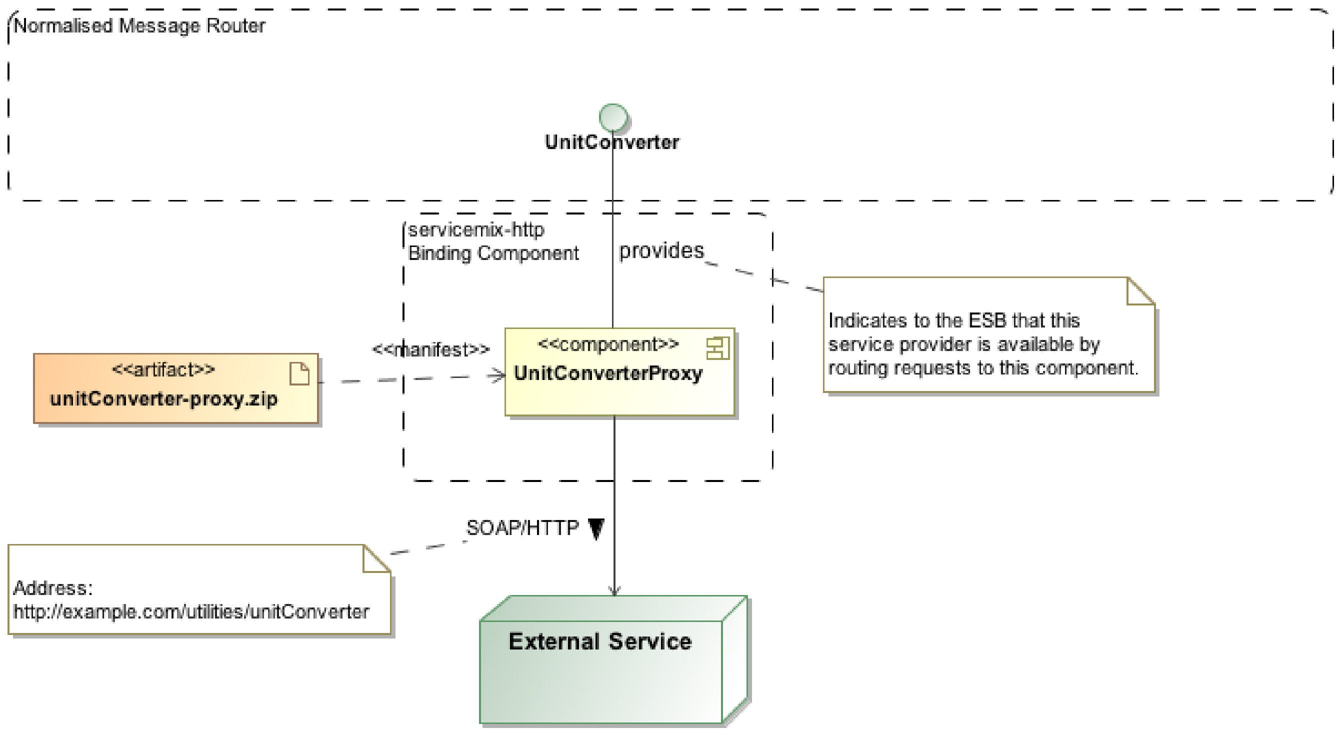

Normalised Message Router
UnitConverter
servicemix-http
Binding Component
provides
Indicates to the ESB that this service provider is available by routing requests to this component.
<<artifact>>
unitConverter-proxy.zip
<<manifest>>
<<component>>
UnitConverterProxy
SOAP/HTTP
Address:
http://example.com/utilities/unitConverter
External Service

Figure 4.17: JBI Example: Publishing an external service to the ESB

We need to

- Create a high-level Maven project (with a `pom.xml`) describing our project
- Create a service unit which we will deploy to the `servicemix-http` binding component. We can do so via the Maven archetype `servicemix-http-provider-service-unit`.
- Modify the resource which is deployed to the binding component (`xbean.xml`) to indicate that it must expose a *provider* endpoint, and tell it which service contract (interface name) it provides to our infrastructure.
- Create a *service assembly* module (via Maven's `servicemix-service-assembly` archetype) to which we add, as dependencies, the above service unit.

Our Maven project structure is as follows:

```
publishExternalServiceToBus/
    pom.xml
    publishToExternalService-sa/
        pom.xml
    unitConverter-proxy/
        pom.xml
        src/
            main/
                resources/
                    xbean.xml
```

The high-level project's POM:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
```

```
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>za.co.solms.examples</groupId>
    <artifactId>publishExternalServiceToBus</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <name>Example: Publish external service to ESB</name>

  <!-- The modules that form part of this composite project -->
  <modules>
    <module>unitConverter-proxy</module>
    <module>publishToExternalService-sa</module>
  </modules>

  <!-- General (global) configuration properties, used by plugins etc -->
  <properties>
    <servicemix-version>3.2.3</servicemix-version>
  </properties>

</project>
```

4.1.8.1.1 Modules

4.1.8.1.1.1 unitConverter-proxy

The service unit’s POM (generated by Maven):

```
                    <?xml version="1.0" encoding="UTF-8"?>
<project>
  <parent>
    <artifactId>publishExternalServiceToBus</artifactId>
    <groupId>za.co.solms.examples</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <modelVersion>4.0.0</modelVersion>
  <artifactId>unitConverter-proxy</artifactId>
  <packaging>jbi-service-unit</packaging>
  <name>A Proxy to an external service provider (UnitConverter)</name>
  <version>1.0-SNAPSHOT</version>

  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <includes>
          <include>**/*</include>
        </includes>
      </resource>
    </resources>
    <plugins>
      <plugin>
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <version>${servicemix-version}</version>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.apache.servicemix</groupId>
      <artifactId>servicemix-http</artifactId>
      <version>${servicemix-version}</version>
    </dependency>
  </dependencies>

</project>
```

The deployed artifact, `xbean.xml`, indicates to the HTTP binding component that we wish to plug in an external service, implementing the `UnitConverter` interface (as per the unit converter's WSDL) running at a particular location.

```
                    <?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:http="http://servicemix.apache.org/http/1.0"
       xmlns:uc="http://example.co.za/unitconverter/"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/http/1.0
       http://servicemix.apache.org/schema/servicemix-http-3.2.2.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <!--  Declare an external service provider, currently running at
        'http://localhost:8080/UnitConverterBeanService/UnitConverter'
        which offers services as per the contract 'uc:UnitConverter'
  -->
  <http:endpoint interfaceName="uc:UnitConverter"
                 service="uc:UnitConverterProxy"
                 endpoint="UnitConverterProxy"
                 role="provider"
                 locationURI="http://localhost:8080/UnitConverterBeanService/UnitConverter"
                 defaultMep="http://www.w3.org/2004/08/wsdl/in-out"
                 soap="true"
                 soapVersion="1.1"/>

</beans>
```

Note
The endpoint and service names make up the logical name *of this proxy endpoint we are deploying*, which could be used to hard-wire other services on the ESB to. It is purely an internal name, and bears no relation to the service/endpoint names declared in the WSDL of the external service

4.1.8.1.1.2 publishToExternalService-sa

The service assembly simply contains a POM (generate by Maven) which we update to refer to the modules which we wish to include in this service assembly. This is done in the *dependencies* section. The `pom.xml`:

```
                    <?xml version="1.0" encoding="UTF-8"?><project>
  <parent>
    <artifactId>publishExternalServiceToBus</artifactId>
    <groupId>za.co.solms.examples</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <modelVersion>4.0.0</modelVersion>
  <artifactId>publishToExternalService-sa</artifactId>
  <packaging>jbi-service-assembly</packaging>
  <name>Service assembly: Example of publishing external service to ESB</name>
  <version>1.0-SNAPSHOT</version>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <version>${servicemix-version}</version>
        <extensions>true</extensions>
        <configuration>
          <type>service-assembly</type>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>

    <!-- Here we list all the modules which must be included in the
    Service Assembly -->
    <dependency>
      <groupId>za.co.solms.examples</groupId>
      <artifactId>unitConverter-proxy</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>

  </dependencies>

</project>
```

Upon packaging, the necessary `jbi.xml` file will be generated by the servicemix / maven tooling, to indicate which service unit(s) - only one in this project - should go to which service engine(s).

4.1.8.1.2 Building and Deploying

Running `mvn install` on the top-level project builds all sub-components, and installs them in the local maven repository.

The service assembly can be deployed, either by running the Maven `jbi:projectDeploy` goal in the service assembly's directory, or simply by copying the service assembly `publishToExternalService-sa-1.0-SNAPSHOT.jar` from the service assembly's `target` directory to your ESB's hot deploy directory.

Depending on the ESB, the service assembly may or not be automatically started. If not, your ESB administration should be used (via Ant, JMX, or vendor-specific tool) to start the assembly.

4.1.8.2 Requesting an external service via the ESB

Consider the scenario where we wish to use the ESB purely as a mechanism to de-couple clients (perhaps stand-alone clients deployed in our organisation) from an external service (a `UnitConverter`) which we published to our ESB earlier via a service provider proxy deployed to the HTTP binding component.

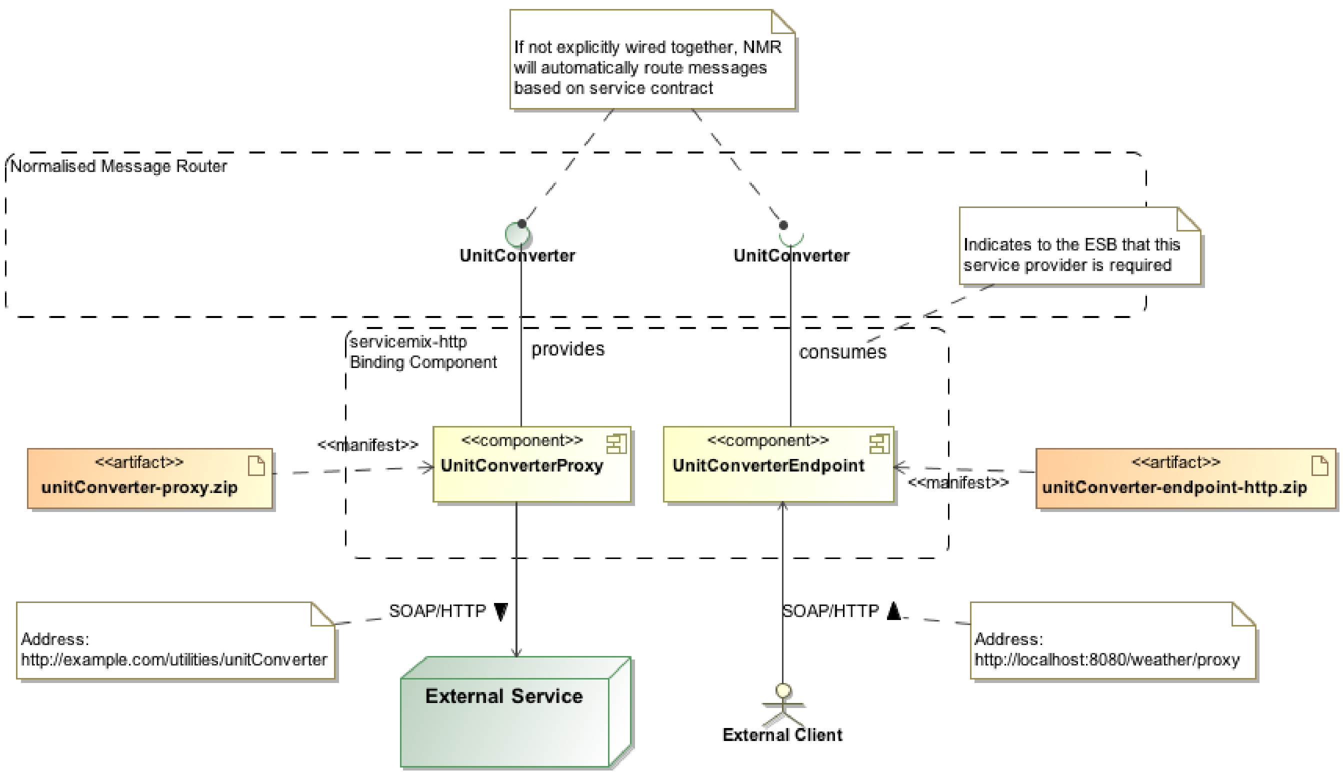

If not explicitly wired together, NMR will automatically route messages based on service contract
Normalised Message Router
UnitConverter
UnitConverter
Indicates to the ESB that this service provider is required
servicemix-http Binding Component
provides
consumes
<<manifest>>
<<artifact>>
unitConverter-proxy.zip
<<component>>
UnitConverterProxy
<<component>>
UnitConverterEndpoint
<<manifest>>
<<artifact>>
unitConverter-endpoint-http.zip
SOAP/HTTP
SOAP/HTTP
Address: http://example.com/utilities/unitConverter
Address: http://localhost:8080/weather/proxy
External Service
External Client

Figure 4.18: JBI Example: Requesting an external service via the ESB

We need to

- Create a high-level Maven project (with a `pom.xml`) describing our project
- Create a service unit (which exposes a HTTP endpoint for clients to call) which we will deploy to the `servicemix-http` binding component. We can do so via the Maven archetype `servicemix-http-consumer-service-unit`.
- Edit the resource which is deployed to the binding component (`xbean.xml`) to indicate that it must expose a *consumer* endpoint (i.e. start a web server), and tell it which service contract (interface name) it consumes. It will thus receive messages via SOAP, and place it on the ESB, which will then be routed to whichever service implements the indicated interface (contract).
- Create a *service assembly* module (via Maven's `servicemix-service-assembly` archetype) to which we add, as dependencies, the above service unit.

Our Maven project structure is as follows:

```
requestServiceViaBus/
    pom.xml
    requestServiceViaBus-sa/
        pom.xml
    unitConverter-endpoint-http/
        pom.xml
        src/
            main/
                resources/
                    xbean.xml
```

The high-level project's POM:

```
            <?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/ ←
    XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0    http:// ←
    maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>za.co.solms.examples</groupId>
    <artifactId>requestServiceViaBus</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <name>Example: Request external service via ESB</name>

  <modules>
    <module>requestServiceViaBus-sa</module>
    <module>unitConverter-endpoint-http</module>
  </modules>

  <properties>
    <servicemix-version>3.2.2</servicemix-version>
  </properties>

</project>
```

4.1.8.2.1 Modules

4.1.8.2.1.1 unitConverter-endpoint-http

The service unit's POM (generated by Maven) for our HTTP consumer (listener) is as follows:

```
                    <?xml version="1.0" encoding="UTF-8"?><project>
  <parent>
    <artifactId>requestServiceViaBus</artifactId>
    <groupId>za.co.solms.examples</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <modelVersion>4.0.0</modelVersion>
  <artifactId>unitConverter-endpoint-http</artifactId>
  <packaging>jbi-service-unit</packaging>
  <name>A Service HTTP Service Engine Service Unit</name>
  <version>1.0-SNAPSHOT</version>

  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <includes>
          <include>**/*</include>
        </includes>
      </resource>
    </resources>
    <plugins>
      <plugin>
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <version>${servicemix-version}</version>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.apache.servicemix</groupId>
      <artifactId>servicemix-http</artifactId>
      <version>${servicemix-version}</version>
    </dependency>
  </dependencies>

</project>
```

The deployed artifact, `xbean.xml`, indicates to the HTTP binding component that we wish to expose (at a specified URL) a SOAP endpoint which receives `UnitConverter` messages and places them on the ESB for routing to any service which implements that contract.

```
                    <?xml version="1.0" encoding="UTF-8"?>

<beans xmlns:http="http://servicemix.apache.org/http/1.0"
       xmlns:uc="http://example.co.za/unitconverter/"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/http/1.0
       http://servicemix.apache.org/schema/servicemix-http-3.2.2.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <!--  Expose a SOAP endpoint which listens for messages
      at 'locationURI', and forwards them to the indicated
      Unit Converter endpoint -->
  <http:endpoint interfaceName="uc:UnitConverter"
                 service="uc:UnitConverterEndpoint"
                 endpoint="UnitConverterEndpoint"
                 role="consumer"
                 locationURI="http://localhost:8192/utilities/unitConverter"
                 defaultMep="http://www.w3.org/2004/08/wsdl/in-out"
                 soap="true" />

</beans>
```

Note
The endpoint and service names make up the logical name *of this consumer endpoint we are deploying*. It is purely an internal name, and bears no relation to the service/endpoint names declared in the WSDL of the external service

4.1.8.2.1.2 requestServiceViaBus-sa

The service assembly simply contains a POM (generate by Maven) which we update to refer to the modules which we wish to include in this service assembly. This is done in the *dependencies* section. The `pom.xml`:

```
                    <?xml version="1.0" encoding="UTF-8"?>
<project>
  <parent>
    <artifactId>requestServiceViaBus</artifactId>
    <groupId>za.co.solms.examples</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>requestServiceViaBus-sa</artifactId>
  <packaging>jbi-service-assembly</packaging>
  <version>1.0-SNAPSHOT</version>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <version>${servicemix-version}</version>
        <extensions>true</extensions>
        <configuration>
          <type>service-assembly</type>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>

    <!-- Add all service units which must form part of the service
      assembly as dependencies -->
    <dependency>
      <groupId>za.co.solms.examples</groupId>
      <artifactId>unitConverter-endpoint-http</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>

  </dependencies>
  <properties>
    <servicemix-version>3.2.2</servicemix-version>
  </properties>
</project>
```

4.1.8.2.2 Building and Deploying

Running `mvn install` on the top-level project builds al sub-components, and installs them in the local maven repository.

The service assembly can be deployed, either by running the Maven `jbi:projectDeploy` goal in the service assembly's directory, or simply by copying the service assembly `requestServiceViaBus-sa-1.0-SNAPSHOT.jar` from the service assembly's `target` directory to your ESB's hot deploy directory.

Depending on the ESB, the service assembly may or not be automatically started, which means the ESB administration should be used (via Ant, JMX, or vendor tool) to start the assembly.

4.1.8.2.3 Testing the service

Using your favourite web services testing tool, we may now send a request message to the URL exposed by our consumer endpoint (`http://localhost:8192/utilities/unitConverter`):

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <convert xmlns="http://example.co.za/unitconverter/">
            <unitConversionRequest>
            <from xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Distance" ↩
                unit="mile">
                    <magnitude>60</magnitude>
                </from>
                <to>kilometer</to>
            </unitConversionRequest>
        </convert>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

which is routed to the external service provider by the ESB, and to which we receive a reply. We may now in the future replace the service provider with a different provider (external, or our own implementation), without affecting clients. This illustrates the value of using the ESB as a mechanism to de-couple clients and service providers, contributing considerably to organisational agility.

4.1.8.3 A Simple Service Provider (Java-Based)

Once we have a services contract, we may decide that we wish to build an implementation of the contract as a service hosted within one a service engine (which itself is hosted by the ESB). There are typically various implementation technologies to choose from, and in this case we build a simple implementation using Java.

The service which we wish to build, is a simple value estimator of motor vehicles. We will deploy it into the `servicemix--cxf-se` service engine, a simple Spring-based service engine that hosts Java classes, and makes their services available on the ESB.

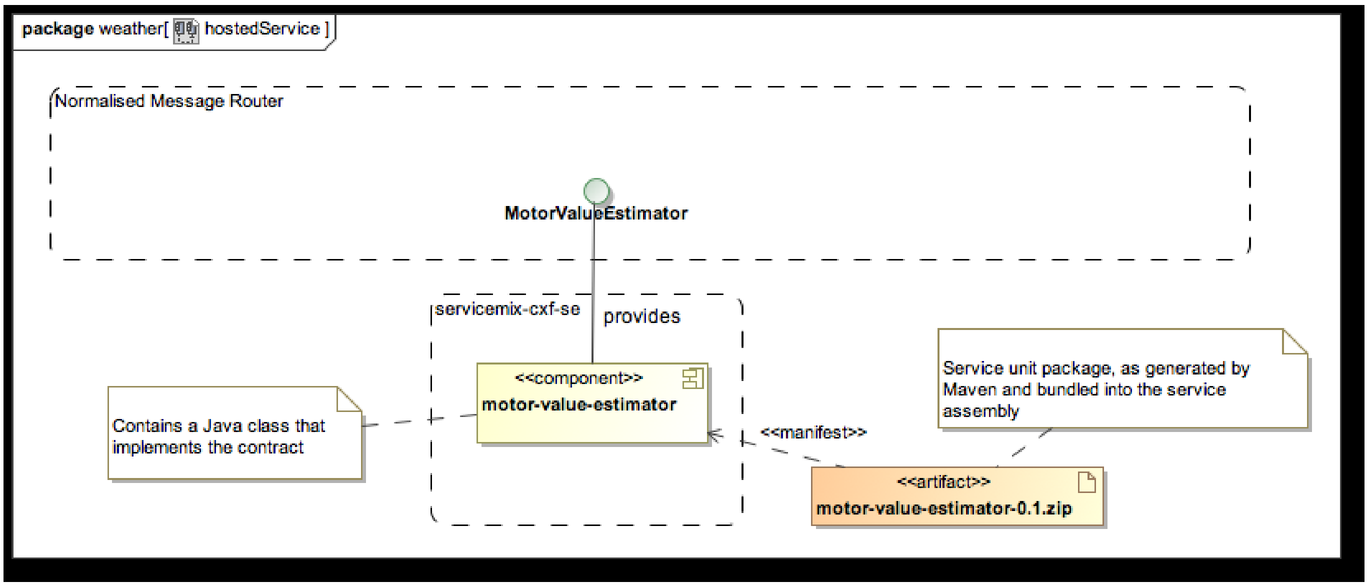

package weather[hostedService]
Normalised Message Router
MotorValueEstimator
servicemix-cxf-se
provides
<<component>>
motor-value-estimator
Contains a Java class that implements the contract
Service unit package, as generated by Maven and bundled into the service assembly
<<manifest>>
<<artifact>>
motor-value-estimator-0.1.zip

Figure 4.19: JBI Example: Building a simple Java-based service

We need to

- Create a high-level Maven project (with a `pom.xml`) describing our project
- Create a service unit (containing a Java class that implements out contract) which we will deploy to the `servicemix-cxf--se` service engine. We can do so via the Maven archetype `servicemix-cxf-se-service-unit`.
- Place our existing WSDL contract (for the value estimator) into the service unit, and use Maven to generate the appropriate Java artifacts
- Edit the `xbean.xml` service unit configuration, to tell it which class(es) to expose to the ESB, and which service and endpoint names to use, etc.

- Create a *service assembly* module (via Maven's `servicemix-service-assembly` archetype) to which we add, as dependencies, the above service unit.

Our Maven project structure is as follows:

```
javaBasedServiceProvider/
    pom.xml
    javaBasedServiceProvider-assembly/
        pom.xml
    motor-value-estimator/
        pom.xml
        src/
            main/
                resources/
                    motor-value-estimator.wsdl
                    motor-value-estimator-soap.wsdl
                    xbean.xml
                java/
                  za/
                    co/
                      solms/
                        ...
                          .java
```

The services contract that we wish to implement looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    targetNamespace="http://solms.co.za/example/motors/"
    xmlns:m="http://solms.co.za/example/motors/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/ http://schemas.xmlsoap.org/wsdl/ ←
        wsdl.xsd http://www.w3.org/2001/XMLSchema http://www.w3.org/2001/XMLSchema.xsd">

    <wsdl:types>
        <xs:schema
            targetNamespace="http://solms.co.za/example/motors/"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified">

            <xs:complexType name="MotorValueEstimationRequest">
                <xs:annotation>
                    <xs:documentation>
                        A request to assess the value of a described motor vehicle.
                    </xs:documentation>
                </xs:annotation>
                <xs:sequence>
                    <xs:element name="manufacturer" type="xs:string"/>
                    <xs:element name="model" type="xs:string"/>
                    <xs:element name="yearOfManufacture" type="xs:gYear"/>
                    <xs:element name="condition" type="m:Condition"/>
                </xs:sequence>
            </xs:complexType>
            <xs:element name="motorValueEstimationRequest" type=" ←
                m:MotorValueEstimationRequest"/>

            <xs:simpleType name="Condition">
                <xs:restriction base="xs:string">
                    <xs:enumeration value="LikeNew"/>
                    <xs:enumeration value="Excellent"/>
                    <xs:enumeration value="Fair"/>
                    <xs:enumeration value="Worn"/>
                    <xs:enumeration value="Damaged"/>
                </xs:restriction>
            </xs:simpleType>

            <xs:complexType name="MotorValueEstimationResult">
                <xs:annotation>
                    <xs:documentation>
                        A result specifying the estimated value of a motor vehicle
                    </xs:documentation>
                </xs:annotation>
                <xs:sequence>
                    <xs:element name="estimatedTradeValue" type="m:MonetaryValue"/>
                    <xs:element name="estimatedRetailValue" type="m:MonetaryValue"/>
                </xs:sequence>
                <xs:attribute name="estimationDateTime" type="xs:dateTime" use="required"/>
            </xs:complexType>
            <xs:element name="motorValueEstimationResult" type=" ←
                m:MotorValueEstimationResult"/>

            <xs:complexType name="MonetaryValue">
                <xs:sequence>
                    <xs:element name="amount" type="xs:double"/>
                </xs:sequence>
                <xs:attribute name="currency" type="m:CurrencyCode"/>
            </xs:complexType>

            <xs:simpleType name="CurrencyCode">
                <xs:annotation>
                    <xs:documentation>
                        ISO currency code
                    </xs:documentation>
                </xs:annotation>
                <xs:restriction base="xs:string">
                    <xs:pattern value="[A-Z]{3}"/>
                </xs:restriction>
            </xs:simpleType>

            <xs:complexType name="UnknownMotorVehicle">
                <xs:annotation>
                    <xs:documentation>
                        A fault indicating that a specified motor vehicle represents a type ←
                             or model
                        unknown to the service provider, and thus non-estimable.
                    </xs:documentation>
                </xs:annotation>
            </xs:complexType>
            <xs:element name="unknownMotorVehicle" type="m:UnknownMotorVehicle"/>

        </xs:schema>
    </wsdl:types>


    <wsdl:message name="motorValueEstimationRequest">
        <wsdl:part name="motorValueEstimationRequest" element=" ←
            m:motorValueEstimationRequest"/>
    </wsdl:message>
    <wsdl:message name="motorValueEstimationResult">
        <wsdl:part name="motorValueEstimationResult" element="m:motorValueEstimationResult" ←
            />
    </wsdl:message>
    <wsdl:message name="unknownMotorVehicle">
        <wsdl:part name="unknownMotorVehicle" element="m:unknownMotorVehicle"/>
    </wsdl:message>


    <wsdl:portType name="MotorValueEstimator">

        <wsdl:operation name="estimateMotorValue">
            <wsdl:input message="m:motorValueEstimationRequest"/>
            <wsdl:output message="m:motorValueEstimationResult"/>
            <wsdl:fault name="unknownMotorVehicle" message="m:unknownMotorVehicle"/>
        </wsdl:operation>

    </wsdl:portType>

</wsdl:definitions>
```

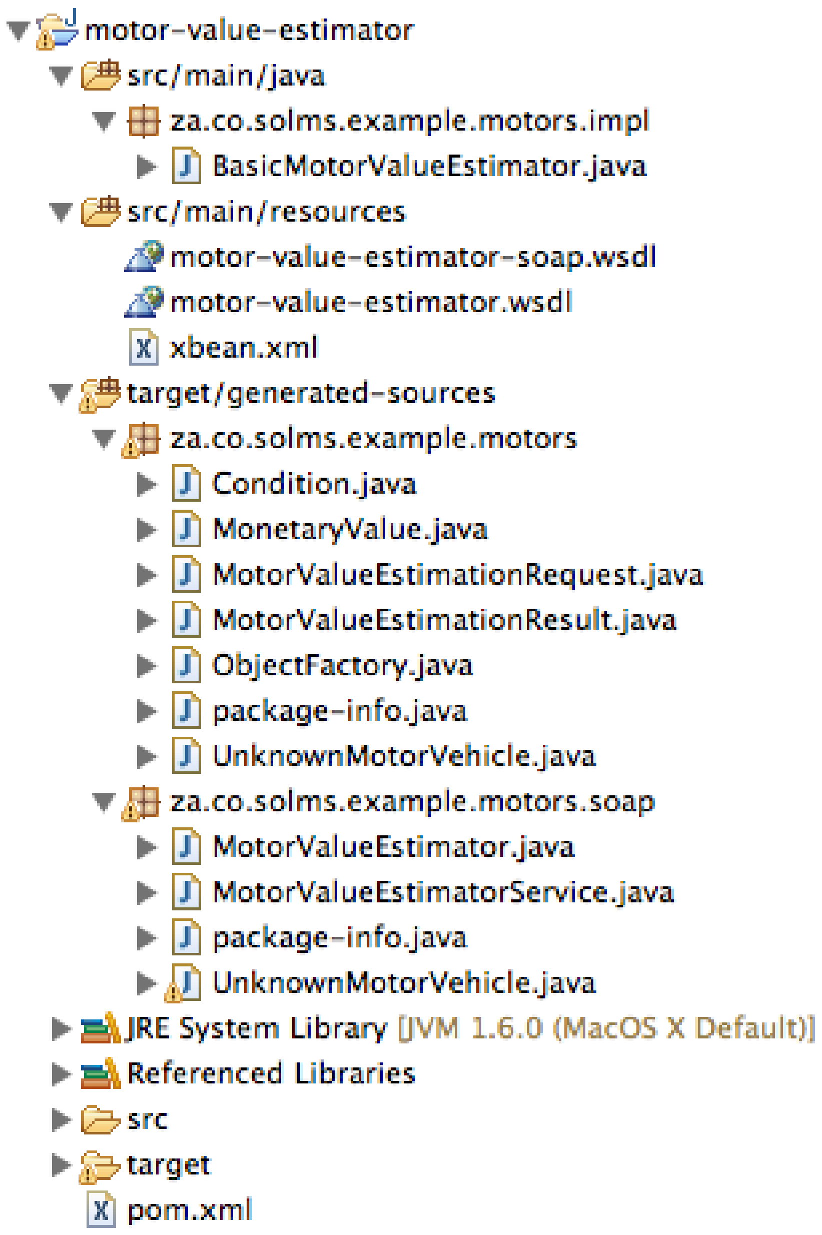
The high-level project's POM:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/ ←
    XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0      http:// ←
    maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>za.co.solms.examples</groupId>
    <artifactId>javaBasedServiceProviderExample</artifactId>
    <version>0.1</version>
    <packaging>pom</packaging>

    <name>Example: A Java-based service provider implementation</name>

  <!-- The modules that form part of this composite project -->
  <modules>
    <module>motor-value-estimator</module>
    <module>javaBasedServiceProvider-assembly</module>
  </modules>

  <!-- General (global) configuration properties, used by plugins etc -->
  <properties>
    <servicemix-version>3.2.3</servicemix-version>
  </properties>

</project>
```

4.1.8.3.1 Modules

4.1.8.3.1.1 motor-value-estimator

This is the service unit that will implement our contract. After placing a copy of the WSDL contract in the resources directory, we generate (using JAX-WS) the java artifacts from the WSDL, and build a simple implementation of the service. The project has the following structure (as seen here in the Eclipse IDE):



```
motor-value-estimator
  src/main/java
    za.co.solms.example.motors.impl
      BasicMotorValueEstimator.java
  src/main/resources
    motor-value-estimator-soap.wsdl
    motor-value-estimator.wsdl
    xbean.xml
  target/generated-sources
    za.co.solms.example.motors
      Condition.java
      MonetaryValue.java
      MotorValueEstimationRequest.java
      MotorValueEstimationResult.java
      ObjectFactory.java
      package-info.java
      UnknownMotorVehicle.java
    za.co.solms.example.motors.soap
      MotorValueEstimator.java
      MotorValueEstimatorService.java
      package-info.java
      UnknownMotorVehicle.java
  JRE System Library [JVM 1.6.0 (MacOS X Default)]
  Referenced Libraries
  src
  target
  pom.xml
```

Figure 4.20: Project Structure of java-based service unit

The service unit's POM is:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/ ↩
   maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>javaBasedServiceProviderExample</artifactId>
    <groupId>za.co.solms.examples</groupId>
    <version>0.1</version>
  </parent>

  <groupId>za.co.solms.examples</groupId>
  <artifactId>motor-value-estimator</artifactId>
```

```
<version>0.1</version>
<packaging>jbi-service-unit</packaging>

<name>A Motor Value Estimator (Java-based)</name>

<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <includes>
        <include>**/*</include>
      </includes>
    </resource>
  </resources>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.servicemix.tooling</groupId>
      <artifactId>jbi-maven-plugin</artifactId>
      <version>${servicemix-version}</version>
      <extensions>true</extensions>
      <configuration>
        <type>service-unit</type>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>jaxws-maven-plugin</artifactId>
      <executions>
        <execution>
          <phase>generate-sources</phase>
          <goals>
            <goal>wsimport</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <target>2.1</target>
        <wsdlUrls>
          <!-- Our contract -->
          <wsdlUrl>${basedir}/src/main/resources/motor-value-estimator-soap.wsdl</wsdlUrl ↩
              >
        </wsdlUrls>
        <sourceDestDir>target/generated-sources</sourceDestDir>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>com.sun.xml.ws</groupId>
          <artifactId>jaxws-tools</artifactId>
          <version>2.1.1</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
</build>
```

```
  <dependencies>
    <!-- We will deploy to CXF Service Engine -->
    <dependency>
      <groupId>org.apache.servicemix</groupId>
      <artifactId>servicemix-cxf-se</artifactId>
      <version>${servicemix-version}</version>
    </dependency>
    <!-- Unit Testing -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>[4.1,]</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <repositories>
    <repository>
      <id>java.net2</id>
      <url>http://download.java.net/maven/2/</url>
    </repository>
  </repositories>

  <pluginRepositories>
    <pluginRepository>
      <id>java.net2</id>
      <url>http://download.java.net/maven/2/</url>
    </pluginRepository>
  </pluginRepositories>

</project>
```

After running Maven's `generate-sources` goal, we can write a simple pure Java implementation of the service provider:

```
package za.co.solms.example.motors.impl;

import java.util.GregorianCalendar;
import javax.jws.WebService;
import javax.xml.datatype.DatatypeFactory;
import za.co.solms.example.motors.*;
import za.co.solms.example.motors.soap.MotorValueEstimator;
import za.co.solms.example.motors.soap.UnknownMotorVehicle;

/** A basic implementation of the motor value estimator contract */

@WebService(endpointInterface="za.co.solms.example.motors.soap.MotorValueEstimator")
public class BasicMotorValueEstimator implements MotorValueEstimator
{
  public MotorValueEstimationResult estimateMotorValue( MotorValueEstimationRequest request ↩
      )
  throws UnknownMotorVehicle
  {
    // Simple bogus implementation of value guessing logic
    double v = Math.random() * 100000.0;

    if (request.getCondition().equals( Condition.LIKE_NEW))
    {
      v *= 1.5;
    }
    else if (request.getCondition().equals( Condition.WORN))
    {
      v *= 0.5;
    }


    MonetaryValue retailValue = new MonetaryValue();
    retailValue.setAmount( v );
    retailValue.setCurrency( "ZAR" );

    MonetaryValue tradeValue = new MonetaryValue();
    tradeValue.setAmount( v * 0.8 ); // 80% of retail
    tradeValue.setCurrency( "ZAR" );

    MotorValueEstimationResult result = new MotorValueEstimationResult();
    result.setEstimatedRetailValue( retailValue );
    result.setEstimatedTradeValue( tradeValue );

    try
    {
      result.setEstimationDateTime( DatatypeFactory.newInstance().newXMLGregorianCalendar( ↩
          new GregorianCalendar() ) );
    }
    catch (Exception e)
    {
      throw new RuntimeException("Failed to set XML date/time", e);
    }

    return result;
  }
}
```

The deployment configuration, `xbean.xml`, indicates to the CXF service engine which java object(s) to create, and how to expose them to the ESB. We can also wire up other ESB-based services in order for our Java object to call them, in this file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:cxfse="http://servicemix.apache.org/cxfse/1.0"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xmlns:m="http://solms.co.za/example/motors/"
       xsi:schemaLocation="http://servicemix.apache.org/cxfse/1.0 http://servicemix.apache. ↩
           org/schema/servicemix-cxfse-3.2.3.xsd
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/ ↩
           beans/spring-beans-2.0.xsd">

    <cxfse:endpoint interfaceName="m:MotorValueEstimator">
        <cxfse:pojo>
            <bean class="za.co.solms.example.motors.impl.BasicMotorValueEstimator" />
        </cxfse:pojo>
    </cxfse:endpoint>

</beans>
```

4.1.8.3.1.2 javaBasedServiceProvider-assembly

The service assembly simply contains a POM (generate by Maven) which we update to refer to the modules which we wish to include in this service assembly. This is done in the *dependencies* section. The `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/ ↩
    maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>javaBasedServiceProviderExample</artifactId>
    <groupId>za.co.solms.examples</groupId>
    <version>0.1</version>
  </parent>
  <groupId>za.co.solms.examples</groupId>
  <artifactId>javaBasedServiceProvider-assembly</artifactId>
  <version>0.1</version>
  <packaging>jbi-service-assembly</packaging>

  <name>Service Assembly</name>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <version>${servicemix-version}</version>
        <extensions>true</extensions>
        <configuration>
          <type>service-assembly</type>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <!-- Modules to include in the assembly -->
    <dependency>
      <groupId>za.co.solms.examples</groupId>
      <artifactId>motor-value-estimator</artifactId>
      <version>0.1</version>
    </dependency>
  </dependencies>

</project>
```

4.1.8.3.2 Building and Deploying

Running `mvn install` on the top-level project builds al sub-components, and installs them in the local maven repository.

The service assembly can be deployed, either by running the Maven `jbi:projectDeploy` goal in the service assembly's directory, or simply by copying the service assembly `javaBasedServiceProvider-assembly-0.1.jar` from the service assembly's `target` directory to your ESB's hot deploy directory.

Depending on the ESB, the service assembly may or not be automatically started, which means the ESB administration should be used (via Ant, JMX, or vendor tool) to start the assembly.

4.1.8.3.3 Testing the service

Since the service is implemented in Java, during development, standard test-driven development practise may be followed.

Once deployed, any pother component (such as an HTTP consumer) may be configured to send messages to an endpoint of type `{http://solms.co.za/example/motors/}MotorValueEstimator` and these should be routed to our implementation.

Chapter 5

Business Process definition and execution

Once the qualities of the ESB (Enterprise Service Bus) become apparent in terms of integrating diverse internal or external services in a de-coupled manner via their contracts, and agnostic to network protocol or implementation technology, it is clear that a significant risk still exists of re-producing a point-to-point integration scenario, where every service would still need to understand a part of the high-level workflow, and (in)directly interact with a potentially large number of other services. By requesting services in a point-to-point manner via the ESB does not solve the inherent design problem.

Consequently, in addition to a contracts-driven de-coupling of services we need *a workflow controller at each level of granularity*, whose sole responsibility is to drive the use-case (service) being offered. The URDAD design process enforces this principle. Such a controller could be implemented using any of the available component technologies which can be hosted in a *Service Engine* connected to an ESB, such as a Java class deployed in a POJO container.

Each such service provider, however, must be *stateless* (as per the SOA definition of a service) and is written to fulfil the client-facing services contract. There is, however, another construct which, from a business perspective, is very pertinent: *the Business Process*.

5.1 What is a Business Process ?

In most implementations (technology- or human-based), a business process is a virtual entity - it is merely a *flow of control*. Through a number of possible triggers (such as receiving a service request from a client) a business process may be started, but when that first service invocation completes, the business process may continue to exist.

Further interactions between the client and different services may change the state of the business process. Consider the act of enrolling for a course at a training provider (a simplification): The entire business process may be seen as one virtual construct, of which the state changes as the client interacts with various different service providers:

- when the enrolments clerk receives an enrolment request, the business process starts, and several activities are performed (such as issuing an invoice for payment)
- three days before the course, the student is asked to confirm his intended attendance,
- on each day of the course, the student confirms attendance with an attendance register,
- on the last day of the course, if the student attended at least 80% of the course, he is issued with a certificate of attendance,
- once the accounting department confirms that the invoice was paid, the business process is complete.

A business process is thus a *flow of control* (maintained by, and only visible to the business) that may transcend individual service invocations and, potentially across different service providers. It may last for a potentially long time, presents a more business-centric view than the (inherently client-focused) view presented by the multiple contracts at multiple levels of granularity, as produced by a design process like URDAD.

5.2 What does a Business Process Execution Engine do?

A business process execution engine is responsible for hosting *instances* of business processes - not as virtual constructs - but as real, stateful artifacts.

Though all service interactions appear to be through various service providers from the client's point of view (each implementing their own contract), the BPEE allows for the process in its entirety to be specified as a single easily-manageable construct, instead of having to fragment the workflow logic across different services, with each knowing it's ‘bit’ of the process, and then having to call the other services for the next process steps.

The BPEE thus directly contributes the maintainability and agility of business processes, by hosting the complete specification of a business process in one artifact. To the client, however, it may still appear is if the business process is a virtual entity; all interactions with the business being performed via multiple different services as before.

5.3 Business Process Execution Engine Implementations

The public standard for specifying processes which are executed in a Business Process Execution Engine is `WS-BPEL`, and a number of engines which can host WS-BPEL processes exist.

Such engines are usually designed to be connected to an enterprise services bus, in order to de-couple the WS-BPEL process from knowing the specifics of the clients and service providers they interact with.

The most prominent standard for ESB components is JBI (Java Business Integration), which enable Business Process Execution Engines (concretely called *WS-BPEL Engines*) to be deployed to any JBI-compliant ESB.

5.4 Typical Business Process Execution Engine features

In order to support useful, centrally-defined business processes, most BPEE implementations should support

- *service orchestration*, the ability to assemble a work flow from existing services. Services should be abstractly referenced (via their contracts) in order to guarantee service plug-ability,
- *long-running business processes*, the state of which is reliably maintained across server software and hardware failures / restarts, necessitating
- the emphasis of *compensating activities* as a replacement for transactions, which only work for very short-lived business processes,
- a mechanism to support the association of incoming messages with a particular (i.e. the client's) instance of a business process
- monitoring and management of business processes, in order to see which instances are active, to cancel processes, and so on.
- the ability to request services from humans, in order to move business processes out of the hands of people, and into a reliable and accessible specification.

A Business Process Execution language should operate in an environment agnostic to messaging protocol / service implementation technology, with protocol adaption services provided by an environment such as an ESB.

5.5 WS-BPEL

5.5.1 Introduction

WS-BPEL, the *Business Process Execution Language*, is an public standard language to declaratively specify executable business processes which are executed in a Business Process Execution Engine.

5.5.1.1 What is BPEL?

WS-BPEL is an XML-based language for the formal specification of business processes and business interaction protocols. WS-BPEL builds upon the Web Services (Abstract WSDL) interaction model, and enables it to support business transactions and work flows.

WS-BPEL evolved from the earlier BPEL4WS (Business Process Execution Language for Web Services) which was handed to OASIS to standardise, which resulted in WS-BPEL version 2.0 (the first official version).

5.5.1.2 Who manages the BPEL specification?

BPEL is the result of a cross-company initiative between IBM, BEA and Microsoft to develop a universally supported process-related language. IBM, Microsoft, SAP, Siebel, BEA and Sun have submitted the BPEL for to certification as a public standard t be managed by OASIS (the Organization for the Advancement of Structured Information Standards) by Oracle, .

5.5.1.2.1 Who is OASIS?

OASIS (the Organization for the Advancement of Structured Information Standards) is a non-profit, international consortium that drives the development, convergence, and adoption of e-business standards.

OASIS has more than 500 member companies and is behind a large number of web specifications including

- ebXML, a huge framework of standards for electronic commerce,
- UDDI, the Universal Description, Discovery and Integration,
- docBook, a xml standard for specifying documents, and
- WS-BPEL, the Business Process Execution Language.

5.5.1.3 BPEL as the heart of the business process

The vision for services oriented architecture is to enable the assembly of business processes from (possibly existing) services which are available across various endpoints (e.g. systems). Using WS-BPEL, these business processes are executed in a business process execution engine - and are defined in a language well suited to service orchestration.

But how does one specify the business process? When designing or communicating the business process, UML (the *Unified Modeling Language*) and/or BPMN (the *Business Process Modeling Notation*) can, and should be used for this. However, these models are typically not *executable*. When following the MDA (Model Driven Architecture), the initial UML model would be technology-neutral, and could be mapped onto a chosen architecture. UML is thus not really suitable for specifying exactly how a business process is to be technically executed within a Service-Oriented Architecture.

Following traditional good design principles, the abstract UML model would have to manually be mapped to an implementation architecture, such as Java EE. But implementing even this well-known mapping is labour-intensive, and requires a high level of developer skill.

In the interest of improving productivity and flexibility, several vendors have thus implemented SOA and/or integration products which enable the rapid assembly and deployment of loosely-coupled workflows. However, not having a public standard for specifying how a business process is to be executed within an environment (typically an enterprise services bus) resulted in the business process execution specification being locked to the ESB vendor, and required that technical experts learn a new language for each new tool.

BPEL aims to address this problem by being a public standard, thus eliminating vendor lock-in by being deployable in several different products and/or environments.

5.5.1.4 WS-BPEL and WSDL

Within WS-BPEL, Web Service Contracts (WSDL) are used for a number of purposes:

- Every BPEL process is itself exposed as a service using on one or more WSDL contracts.
- WSDL is used to describe the public entry and exit points for the process.
- WSDL data types are used within a BPEL process to describe the information that passes between requests.
- WSDL is used to reference external services required by the process.
- The WSDL *partner links* extension is used to indicate a BPEL process' relationship with clients, as well as other services (the *collaboration context* from UML).

5.5.1.5 How is BPEL generated?

BPEL is meant to be generated not only by developers, but directly by business analysts and managers. To this end BPEL would not normally be directly generated via an XML editor, but through some diagrammatic business process design tool which typically uses either

- UML diagrams with potentially some business process extensions,
- or the OMG's Business Process Modeling Notation (BPMN) which includes as part of the language specification a BPMN to BPEL mapping.

In practice, most BPEL modeling tools use their own graphical notation, anf the BPEL developer should be careful not to simply become a 'user' of such a tool. In such a case, it is important to understand the vocabulary and its semantics on their own.

The longer term vision is to generate BPEL just like any other code artifact, as part of a Model-Driven Development process where a technology-neutral UML model (the PIM) is mapped to various technologies based on the architectural choices made for each component.

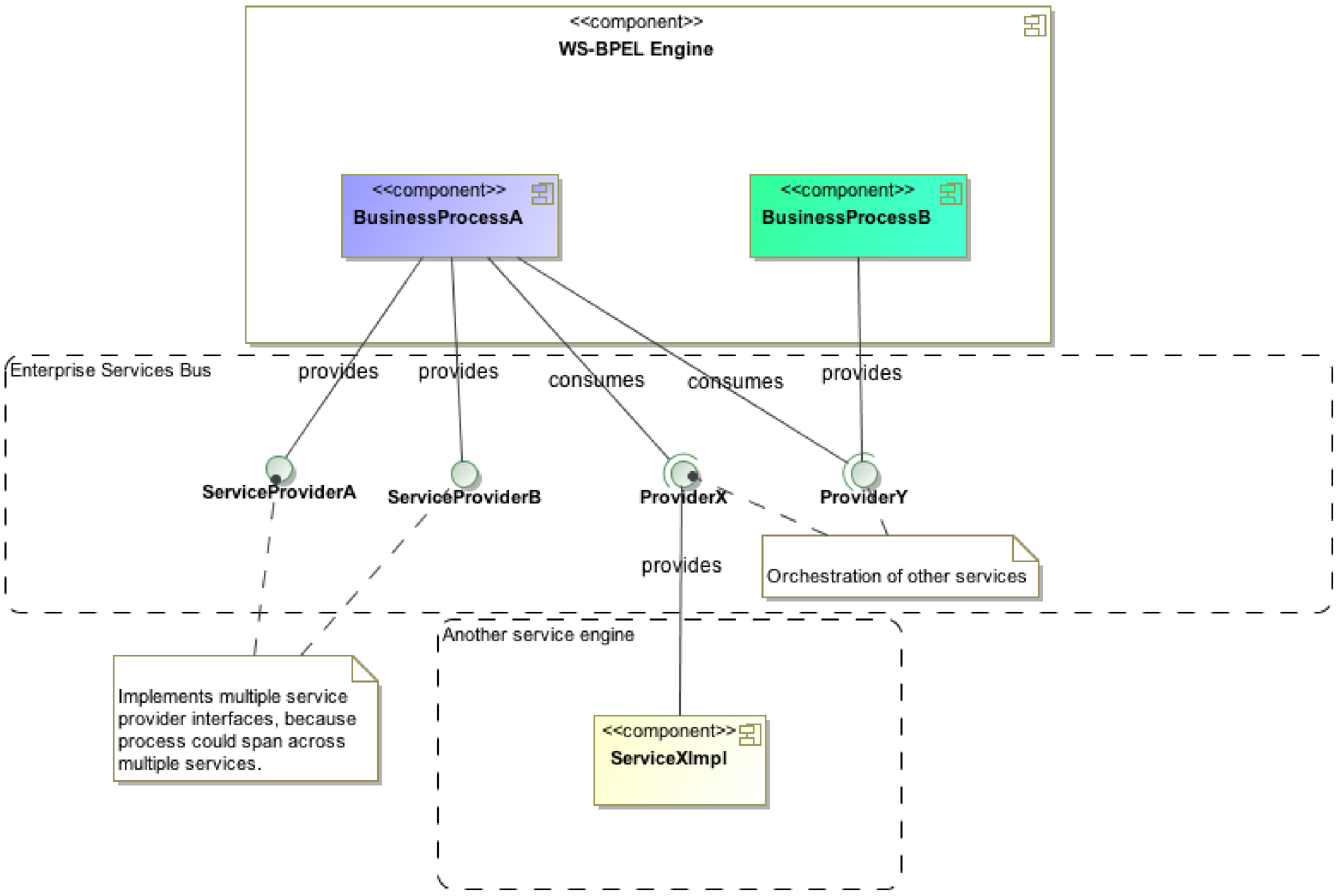


Figure 5.1: BPEL as hosted business processes

5.5.2 Core features of BPEL

Since the concept of a Business Process is not a construct normally provided by a typical object-oriented language (as it is usually an implicit flow of control across objects), BPEL introduces this concept in a manageable way for anybody who wishes to implement services in a process-oriented manner.

In BPEL, a business process is composed of elements ("activities") that define activity behaviours. These include the interactions, workflow control, and managing workflow state and information.

5.5.2.1 BPEL interactions

The BPEL interactions include

- invoking another service via `<invoke>`,
- generating a response for a synchronous service via `<reply>`,
- notifying a client that a requested service is not going to be provided due to either
  - a precondition for the service not having been met (an exception)
  - or due to a system problem preventing the service provider to fulfil its contractual obligations (an error)

  using `<throw>`

- waiting for a client to invoke a service or send a message using `receive`,
- the ability to use message correlation to match *asynchronous* request and response objects with one another.

5.5.2.2 BPEL flow control

The control statements include

- specifying that certain tasks should be executed sequentially via `<sequence>`,
- the certain tasks should be executed concurrently via `<flow>`,
- that certain paths are followed conditionally via `<switch>` or `<pick>`,
- that certain tasks should be executed repetitatively via `<while>`
- that certain transactions should be effectively reversed via inverse or compensating transactions

Note
BPEL supports long running business transactions and activities.

- the ability to `wait` for a certain time within a workflow,
- the ability to handle exceptions and errors via appropriate handler services, and
- the ability to terminate a business process,

5.5.2.3 Managing workflow data

In order to manage ad maintain work flow state information, BPEL supports variables (together with copy and assignment operations) as well as message transformation. For each running business process, the BPEL engine should maintain the process state reliably.

5.5.2.4 Abstract and Executable business processes

The BPEL standard specifies two types of BPEL business processes, each with their own XML schema to assist the developer and lay down the specification of the structure:

- **Abstract processes,** showing only the externally visible interactions of a business process (i.e. the message passing, ordering and timing constraints on interactions between client, process, and/or partner service providers). This unique concept uses abstraction to indicate the general nature of a business process (perhaps to one's client), without divulging the internal implementation details.
- **Executable processes,** The full process (which may *implement* an abstract process) which is deployed to, and executed by, the BPEL engine.

For example, if complex interaction mst occur between a client and the service provider during the business process (imagine, say, a complex legal process) it may be difficult to statically express the full nature of the interaction simply via contracts with pre- ad postconditions.

Instead, the service provider provides the client with the abstract version of the process, showing only the interactions and acting as a 'template' within which the real process must fall. This abstract process may even include so-called opaque extension points, within which the client may perform it's own activities (opaque to the service provider) in the executable business process, but which may not violate the overall process rules. The same, of course, may apply in the inverse scenario where a client expresses, as part of his needs for a service provider, an abstract BPEL process.

This, together with the services contract, moves towards specifying a richer overall contract which may also help greatly in automatically matching clients and servers (determining whether a service provider is suitable to a client's needs).

5.5.2.5 Long-running business processes

If one thinks of a long business process in the real world, as soon as a client interacts with the service provider, the client needs to produce some information in order to allow the service provider to associate the client with a particular business process. This usually occurs implicitly (i.e. some or other piece of information in the message sent from the client to the provider will provide the necessary link), and is not usually a separate service request.

In this spirit, BPEL supports correlation via *correlation sets*, a declarative mechanism to indicate which information in the exchanged messages (one or more *message properties*) is used to identify an existing instance of a business process, instead of creating a new instance which receives the request.

5.5.3 A simple example

Let us have a look at a simple hello-world example to develop a feeling for bpel and its interaction with wsdl definitions.

5.5.3.1 The business process specification

Consider a simple business process where the business process

- receives a synchronous message,
- Constructs a simple response message, and
- sends the response back to the client.

The BPEL file defining the business process is shown below:

```
        <?xml version="1.0" encoding="UTF-8"?>
<process
    name="helloWorld"
    targetNamespace="http://www.mycomp.org/bpel/helloWorld"
    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://www.solms.co.za/wsdl/helloWorld"
    xmlns:bpel="http://www.mycomp.org/bpel/helloWorld"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://docs.oasis-open.org/wsbpel/2.0/process/executable
    http://docs.oasis-open.org/wsbpel/2.0/CS01/process/executable/ws-bpel_executable.xsd">

    <!-- Import contracts and/or schemas, which contain partner link definitions -->
    <import
        namespace="http://www.solms.co.za/bpel/helloWorld"
        location="helloWorld.wsdl"
        importType="http://schemas.xmlsoap.org/wsdl/"/>

    <!-- Specify the role this process lays in each of these partner links -->
    <partnerLinks>
        <partnerLink name="HelloWorldProviderLink"
                     partnerLinkType="wsdl:HelloWorldPartnerLinkType"
                     myRole="helloWorldProvider"/>
    </partnerLinks>

    <!-- The state of the business process -->
    <variables>
        <variable name="helloWorldInput" messageType="wsdl:hello"/>
        <variable name="helloWorldOutput" messageType="wsdl:helloResponse"/>
    </variables>

   <!-- The business process -->
   <sequence>

        <receive name="helloReceipt"
                 partnerLink="HelloWorldProviderLink"
                 portType="wsdl:HelloWorldPortType"
                 operation="helloWorld"
                 variable="helloWorldInput"
                 createInstance="yes"/>

        <assign name="Assign">
            <copy>
                <from>$helloWorldInput.msg/text</from>
                <to>$helloWorldOutput.orgMessage/txt</to>
            </copy>
            <copy>
                <from>Sawubona</from>
                <to>$helloWorldOutput.response/txt</to>
            </copy>
        </assign>

        <reply name="helloWorld.Response"
               partnerLink="HelloWorldProviderLink"
               portType="wsdl:HelloWorldPortType"
               operation="helloWorld"
               variable="helloWorldOutput"/>
   </sequence>

</process>
```

It defines

- a *partnerlink* for the incoming request and the subsequent synchronous response provided by the business process,
- *variables* used to manage the information maintained across the business process (in our case for the input and output messages),
- and the actual *business process* with 3 sequential steps:
  1. The receipt of the incoming request.
  2. The manipulation of the business process data (constructing a response message from the input message).
  3. Returning the response for the business process to the client (as a synchronous response for the original request).

5.5.3.2 The interface definition file (WSDL)

The business process specification specifies the receipt and sending of messages. In WS-BPEL, the communication is done via web services and hence the interfacing definition is provided in the *Web Services Definition Languagle (WSDL)*.

In our case the WSDL specifies a single web service with the corresponding request and response messages offered through a single port:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="helloWorld"
     targetNamespace="http://www.solms.co.za/wsdl/helloWorld"
     xmlns:tns="http://www.solms.co.za/wsdl/helloWorld"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:plnk="http://schemas.xmlsoap.org/ws/2004/03/partner-link/">

    <wsdl:types>
        <xsd:schema targetNamespace="http://www.solms.co.za/bpel/helloWorld">
            <xsd:element name="textItem">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element type="xsd:string" name="text"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </wsdl:types>

    <wsdl:message name="hello">
        <wsdl:part name="msg" element="tns:textItem"/>
    </wsdl:message>

    <wsdl:message name="helloResponse">
        <wsdl:part name="orgMessage" element="tns:textItem"/>
        <wsdl:part name="response" element="tns:textItem"/>
    </wsdl:message>

    <wsdl:portType name="HelloWorldPortType">
        <wsdl:operation name="helloWorld">
            <wsdl:input message="tns:hello"/>
            <wsdl:output message="tns:helloResponse"/>
        </wsdl:operation>
    </wsdl:portType>

    <binding name="HelloWorldSoapBinding" type="tns:HelloWorldPortType">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="helloWorld">
            <soap:operation soapAction="helloWorld"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </binding>

    <wsdl:service name="helloWorldService">
        <wsdl:port name="helloWorldPort" binding="tns:HelloWorldSoapBinding">
            <soap:address location="http://localhost:12321/helloWorld"/>
        </wsdl:port>
    </wsdl:service>

    <plnk:partnerLinkType name="HelloWorldPartnerLinkType"
              xmlns:plnk="http://schemas.xmlsoap.org/ws/2004/03/partner-link/">
        <plnk:role name="helloWorldProvider" portType="tns:HelloWorldPortType"/>
    </plnk:partnerLinkType>

</wsdl:definitions>
```

Note

The binding and service definitions are often omitted in the WSDL specification for a BPEL defined business process specification as these are often automatically generated by the BPEL execution environment.

5.5.4 The BPEL language

In this section we look at the various elements of the BPEL language.

5.5.4.1 The process root

The root of any WS-BPEL business process specification is a `process` element defined within the WS-BPEL name space:

```
http://docs.oasis-open.org/wsbpel/2.0/process/executable
```

The schema for WS-BPEL Executable, version 2.0 (the first official version) is published at:

```
http://docs.oasis-open.org/wsbpel/2.0/CS01/process/executable/ws-bpel_executable.xsd
```

The structure of a process specification is shown below:

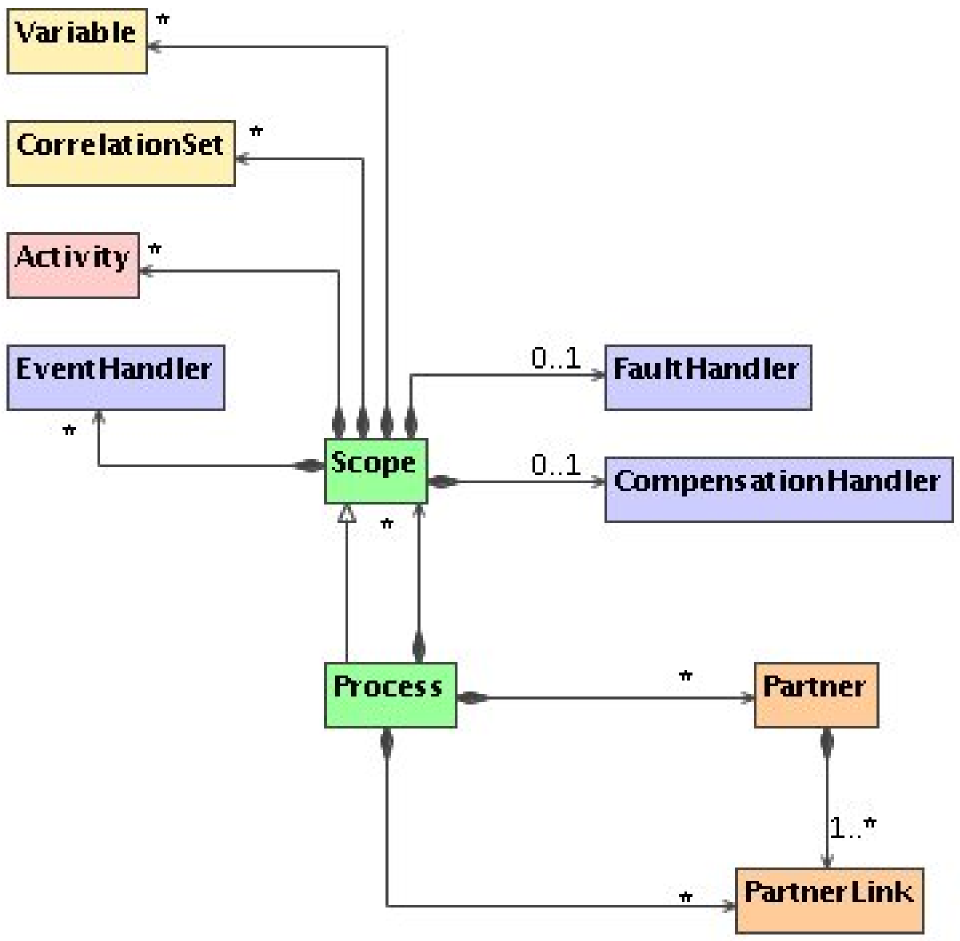


Figure 5.2: The high level structure of a WS-BPEL process

A process is a scope within which activities occur and within which information is maintained. Every process is given a name which typically is related to the use case which is realized in that business process. The elements defined within a BPEL process are thus scoped to within that process. The process defines

- partners and partner links through which the process interacts with the different role players in the business process,
- variables used to maintain state information on the process as well as to construct data objects exchanged with partners,
- correlation sets used to couple asynchronous requests with their corresponding responses,
- a collection of activities which may themselves be structured activities (e.g. sequential or concurrent flow) or basic activities like receiving a messages, assigning variables, invoking requests, throwing faults, ...
- various types of events which result in a different flow in the business process,
- optionally a fault handler for the scope, and
- optionally a compensation handler (for executing inverse transactions which effectively reverse aspects of the workflow which are not reversed automatically within the transaction management context).

A business process may have several *sub-scopes* which localize variables and definitions to within a particular region of the business process.

A typical BPEL process would have the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<process
    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    targetNamespace="http://my.proces/target/namespace"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://docs.oasis-open.org/wsbpel/2.0/process/executable
    http://docs.oasis-open.org/wsbpel/2.0/CS01/process/executable/ws-bpel_executable.xsd"
    name="myProcess">

    <!-- Import partner links definition, service contracts, schema, etc -->
    <import importType="..." location="..."/>
    ...

    <!-- What role do we play in the partnership ? -->
    <partnerLinks>
        <partnerLink name="..." partnerLinkType="..." myRole="..."/>
        ...
    </partnerLinks>

    <!-- Global variables / 'process state' -->
    <variables>
        <variable name="..." messageType="..."/>
        ...
    </variables>

    <!-- Main process flow -->
    <sequence>
        <!-- Receive request -->
        <receive partnerLink="..." operation="..." variable="..." createInstance="yes"/>
        ...
    </sequence>

</process>
```

To illustrate, the interfacing with the partners, look at the sketch of a business process for an insurance claim shown below.

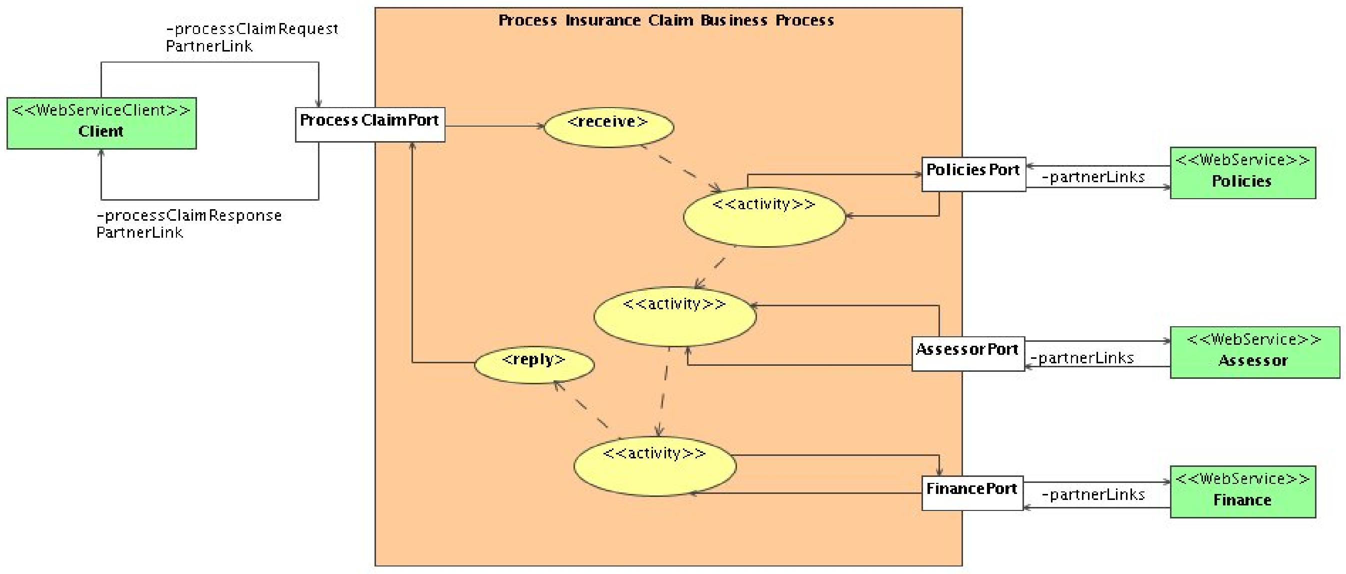


Figure 5.3: The high level structure of a BPEL process

5.5.4.2 Partner links

Partner links are concrete references to services that the business process interacts with. The partner link can specify a service which

- requests the use case realized by the business process,
- a service which receives the response from the business process,
- a lower level service used by the business process as one of the business process steps

5.5.4.2.1 Partner link type specification in the WSDL file

The WSDL defines the *partner link types* which, in turn, defines

- the roles for the partner links together with
- the port types used by these roles

without binding the role to any particular role player in the business process.

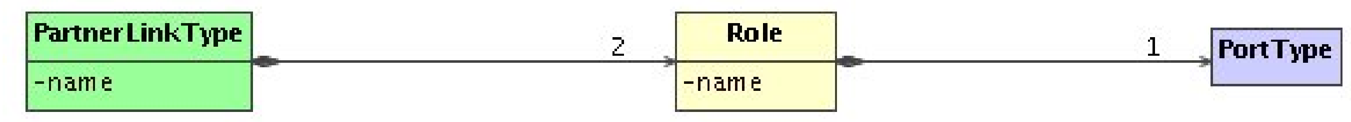


Figure 5.4: The structure of a partner link type

For example, a WSDL is written which imports each of the services contracts of the service providers, and which then establishes the *context of collaboration* by, for each association between any two providers, declaring a partner link:

```
<?xml version="1.0"?>
<definitions
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="..."
    xmlns:ass="http://big-insurance-company.com/assessment/service/"
    xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype">


    <import location="assessor.wsdl" namespace="http://big-insurance-company.com/assessment ←
        /service/"/>


  <!-- Describes the relationship between a client (which
       implements AssementResponseCallBack) and an Assessor -->
  <plink:partnerLinkType name="AssessorLinkType">
    <plink:role name="assessorService">
          <plink:portType name="ass:Assessor"/>
    </plink:role>
     <plink:role name="assessmentRequester">
          <plink:portType name="ass:AssementResponseCallBack"/>
     </plink:role>
  </plink:partnerLinkType>


</definitions>
```

Each `partnerLinkType` is analogous to a UML association relationship between two service providers. It may contain either *one* or *two* roles. Two roles mean that the partner link implies a bi-directional relationship, where both services may at any time call one another.

Most simple services, however, involve unidirectional association, where a client simply calls a service provider, which has no way of calling back the client after a service has completed. This is modelled by including only one role in the partner link.

5.5.4.2.2 Partnerlink specification in the BPEL file

In the business process specification we bind the roles to actual services. This is done at the beginning of the BPEL within a `<partnerLinks>` block which defines the various partner links used in the business process.

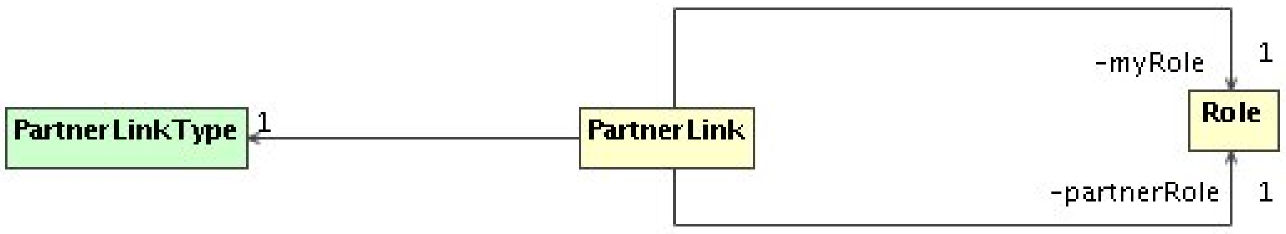


Figure 5.5: The structure of a partner link type

Each partner link

- refers to a partnerlink type declarationm and specifies
- the role played by the business process, and
- the role played by the partner in the context of this partner link.

For example, the roles around the claim assessment defined in the WSDL could now be assigned to specific services via

```
<partnerLinks>

  <!-- The business process indicates which roles it play(s)
       in the various partner links within it participates -->
  <partnerLink
      name="assessment"
      partnerLinkType="wsdl:AssessorLinkType"
      myRole="assessmentRequester"
      partnerRole="assessorService"/>

  <partnerLink .../>

</partnerLinks>
```

5.5.4.3 Defining variables

Variables are used to

- construct parameters used in a service request to a business partner,
- store the response from a business partner for later usage, and
- maintain the state of the business process.

5.5.4.3.1 Variable types

A BPEL variable can hold

- a WSDL message (messageType),
- an element defined in an XML schema (element), or
- a XML simple type (type).

5.5.4.3.2 Declaring a variable

Variables are declared in a `<variables>` block:

```
<process>

  <partnerLinks>
     ...
  </partnerLinks>

  <variables>
      <variable .../>
       ...
      <variable .../>
   </variables>
</process>
```

Each variable is declared by specifying a type and giving the variable a name:

```
<variable name="processClaimRequest" type="wsdl:SubmitInsuranceClaimMessage"/>

<variable name="claim" type="ins:Claim"/>

<variable name="date" type="xsd:date"/>
```

5.5.4.4 WS-BPEL activities

Activities are used to define workflow steps. They include

- activities of interfacing with partners,
- manipulating variables, and
- controlling the flow of the business process.

BPEL defines a range of basic atomic activities as well as a range structured activities which are themselves composed of basic activities.

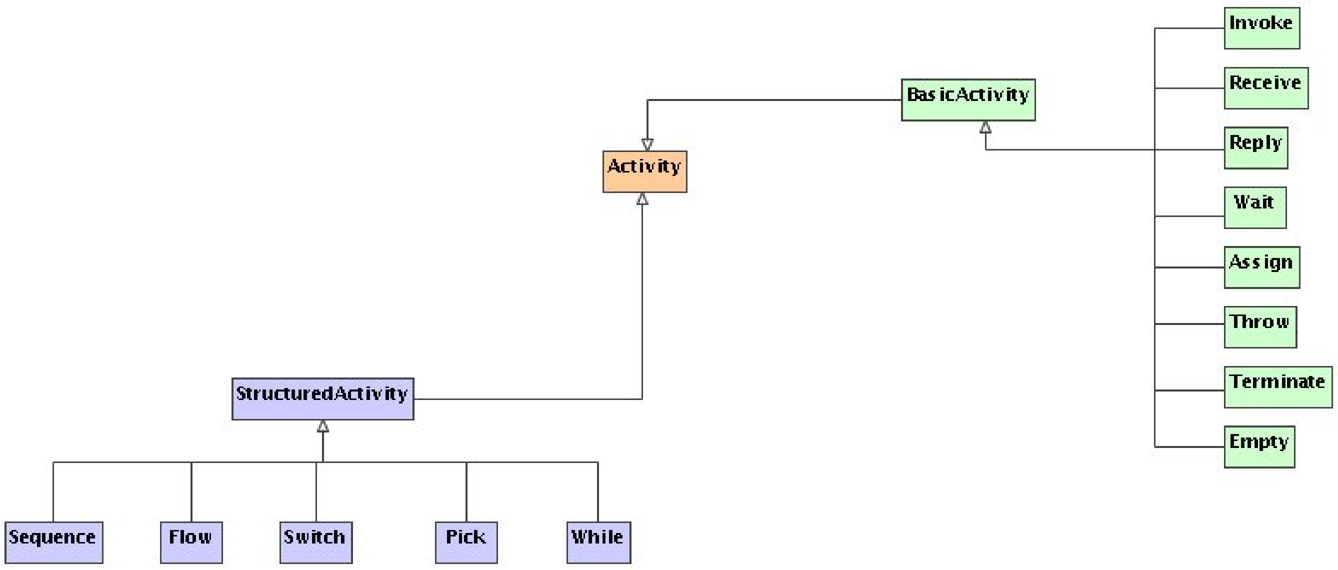


Figure 5.6: BPEL activities

5.5.4.4.1 Basic activities

Basic activities are not composed of further activities.

5.5.4.4.1.1 Invoke

Invoke requests a service from a business partner. Invocations may be synchronous or asynchronous.

For an invocation one specifies

- the partner link to be used,
- the port type to be used,
- the operation to be invoked,
- optionally a variable which contains the request message, and
- and optionally a variable which will be populated with the response message (in the case of synchronous invocations).

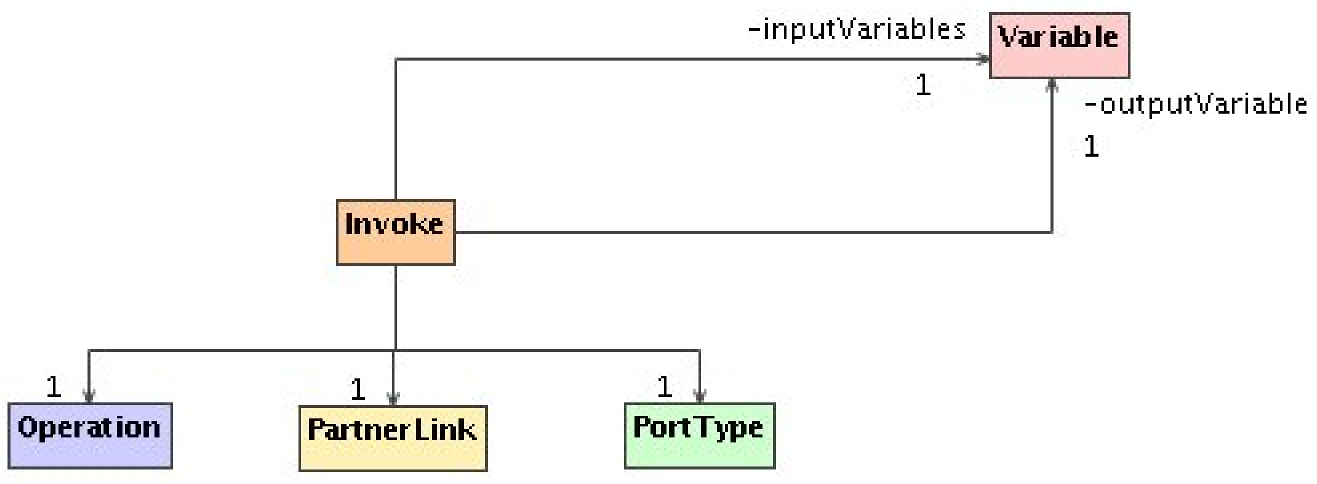


Figure 5.7: The structure of the invoke activity

As an example, let us look at invoking an assessment:

```
<invoke partnerLink="AssessorPartnerLink"
      portType="AssessmentPortType"
      operation="assessClaimValue"
      inputVariable="assessmentRequest"
      outputVariable="assessment"/>
```

5.5.4.4.1.2 Receive

Receive waits for an incoming message before continuing with the business process. It is used to

- receive the initial request which starts the business process, and
- wait from responses from asynchronous requests made to partners of the business process.

The structure of a receive message is illustrated in the following:

```
<receive partnerLink="ClientLink"
      portType="ClaimProcessingPortType"
      operation="processClaim"
      variable="claim"
      createInstance="yes"/>
```

The `createInstance` attribute instructs the BPEL engine to create a new instance of the business process. It is usually set to `true` for the initial request starting the business process and false for receipts of responses to asynchronous requests.

5.5.4.4.1.3 Reply

Reply is used to send a response for a synchronous request. It is always used in conjunction with a `receive`.

The structure of a reply message is shown below:

```
<reply partnerLink="ClientLink"
      portType="ClaimProcessingPortType"
      operation="processClaim"
      variable="AssessmentResponse"/>
```

5.5.4.4.1.4 Variable assignment

BPEL supports the copying and construction of XML based data types via assignment statements. One can either

- assign a complete data structure by initializing it from another variable or aspects from another variable, or one can
- modify aspects of a variable with an assignment to aspects of another variable or to constant values.

5.5.4.4.1.5 General structure of an assignment statement

The assignment statement always copies from some source data into the data structure of a variable. Within an assignment statement there may be multiple copy instructions. The general form is

```
<assign>
    <copy>
        <from .../>
        <to .../>
    </copy>
    <copy>
        <from .../>
        <to .../>
    </copy>
</assign>
```

In its simplest form we simply copy the contents of one variable into the data structure of another variable:

```
<assign>
  <from variable="assessmentResponse"/>
  <to variable="selectedAssessment"/>
</assign>
```

5.5.4.4.1.6 Copying aspects from a structure

Commonly one may want to copy an element from a larger data structure into another variable.

For example, if we have the following data structure

```
<complexType name="Claim">
  <element name="policyNumber" type="xs:string"/>
  <element name="claimItem" type="ClaimItem" maxOccurs="unbounded"/>
  ...
</complexType>
```

and we have a variable, `claim`, of that type, then we could extract, for example, the policy number into another variable

```
<assign>
  <from variable="claim" part="policyNumber"/>
  <to variable="policyId"/>
</assign>
```

5.5.4.4.1.7 Assigning parts of a variable

In a similar way we could assign parts of a variable via, for example

```
<assign>
  <from variable="policyNo"/>
  <to variable="claim" part="policyNumber"/>
</assign>
```

5.5.4.4.1.8 Using a query language for resolving data structure components

the `part` construct provides a very limited way of selecting aspects of a data structure from which data is to be extracted or into which data is to be copied. In general one would one to use a more powerful query language for this purpose.

BPEL enables one to select the query language one would like to use in order to resolve elements of a more complex data structure. The default used by BPEL is `XPath`.

```
<assign>
  <from variable="client"
          part="address"
          query="/address/country"/>
  <to variable="shippingCountry"/>
</assign>
```

5.5.4.4.1.9 Initialization with constant data

One can also assign a BPEL variable to constant XML data. This is illustrated by the following example

```
<assign>
    <copy>
         <from>
            <loc:address xmlns:loc="http://www.solms.co.za/domainObjects/locations">
              <loc:streetAddress> 113 Barry Hertzog Ave</loc:streetAddress>
              <loc:suburb>Emmarentia</loc:suburb>
              <loc:city>Johannesburg</loc:city>
              <loc:country>South Africa</loc:country>
            </loc:address>
        </from>
        <to>
          <variable="ourContactDetails" part="physicalAddress"/>
        </to>
    </copy>
</assign>
```

5.5.4.4.1.10 Delayed execution via wait

At times one wants to delay an activity within a sequence, i.e. to wait for a certain period or until some specified time instant. As BPEL is meant to support long living business processes, the wait periods may, at time, be substantial (e.g. to wait to the end of the month).

5.5.4.4.1.11 Waiting for a specified duration

The `<wait for="duration-expression"/>` element is used to specify for some specific period. The duration is specified as a standard XML schema duration type.

For example, should you wish to wait (for reasons beyond me) for 3 months 2 days 6 hours 12 minutes and 65 seconds before the next activity in the current sequence of activities is to be performed, then you can specify this ass follows:

```
<wait for="P3M2D6H12M65S"/>
```

5.5.4.4.1.12 Waiting until some point in tme

To wait until some point in time, one uses the `<wait until="deadline-expression"`. The deadline is specified as either a XML date or a dateTime as defined in the XML schema data types.

For example, should one wish to wait until 23h00, then one can specify this via

```
<wait until="23:00:00:Z"
```

5.5.4.4.1.13 Immediate process termination via exit

BPEL provides a mechanism to immediately terminate a process without any termination, fault or compensation handlers being executed. You can request immediate termination of a business process via the

```
<exit/>
```

tag.

Note
This may leave the business processes and the system in an invalid state (a state where the business rules or invariance constraints are no longer met) and should not usually be used.

5.5.4.4.1.14 Empty activities

At times one wants to specify an activity which does nothing. This is particularly useful when writing a fault handler which should ignore the fault, such that the business process proceeds as if no fault occured. To this end BPEL defines the `<empty>` activity:

```
<empty/>
```

5.5.4.4.2 Fault/exception signalling and handling

In order to provide a rigid framework for communicating faults up the service hierarchy, WS-BPEL provides an infrastructure for signalling and handling faults. This infrastruture is similar to the standard exception handling infrastructure provided by most modern programming languages.

5.5.4.4.2.1 How does a fault arise?

BPEL defines services as business processes as assembled work flow steps. Each processing step is either

- a finer grained service which in turn is assembled from lower level work flow steps, or
- an atomic activity executed by the bus, or
- an atomic processing step executed by an external service provider.

A fault can arise due to

- not being able to establish communication with a service provider,
- the service provider not being able or willing to provide the requested service,
- the business process rejecting the request message which initiated the service, or
- the business process making a decision to raise a fault based on the current work flow state or based on the response provided from one of the service providers.

5.5.4.4.2.2 Errors versus exceptions

In principle one should distinguish between errors and exceptions where

- an error is a scenario where the service provider does not fulfill its contractual obligations and
- an exception is a scenario where the service provider refuses a service because one or more of the preconditions for that service are not met.

One thus has to consult the contract (interface with preconditions and postconditions and quality requirements) in order to determine whether the non-provision of a service resembles an error or an exceptional situation.

Example of an exception
A bank may provide a debit service subject to there being sufficient funds in the account. Should there be insufficient funds in the account, the bank may refuse the service without braking the contract. This would not be an error as the service provider (the bank) does not violate the contract.

BPEL does not natively differentiate between a exceptions and errors. Either are signalled as faults. Generally one would specify in the WSDL for a service provider

- one fault for each precondition under which the service provider may refuse the service (these are the exceptions), and
- one fault which signals an error, i.e. the inability of the service provider to meet its contractual obligations.

In order to enable the definition of cleaner business processes, one could introduce different base fault types for exceptions and errors.

5.5.4.4.2.3 Fault message declarations

The WSDL for a service provider declares for an operation (service) the input message, output message and any fault messages which may be returned.

```
<portType name="AccountDebitPT">
  <operation name="debit">
    <input message="bank:DebitRequest"/>
    <output message="bank:DebitResponse"/>
    <fault message="bank:InsufficientFundsException"/>
    <fault message="bank:DailyLimitExceededException"/>
    <fault message="bank:DebitError"/>
  </operation>
</portType>
```

5.5.4.4.2.4 Signalling faults

BPEL two ways of communicating faults:

1. On can signal a fault for internal consumption within the business process.
2. One can return a fault message to a client requesting the service.

5.5.4.4.2.5 Signalling faults for internal consumption

Faults are signalled in BPEL via a `<throw>` activity. The tag may specify just the fault type

```
<throw faultName="InsufficientFundsException"/>
```

or may also provide a variable which is resembles the fault message

```
<throw faultName="InsufficientFaultException" faultVariable="brokeException"/>
```

Normally exception signalling is done within a switch statement:

```
<switch>
  <case condition="...">
     <reply .../>
  </case>
  <otherwise>
    <throw faulltName="..." faultVariable="..."/>
  </otherwise>
</switch>
```

5.5.4.4.2.6 Notifying clients of a fault

If the fault is not meant to be processed at higher level services hosted on the bus, then a fault message is returned to the client who requested the high level service. This is done via the normal `<reply>` tag:

```
<reply partnerlink="Client"
                portType="..."
                operation="..."
                variable="myFaultVariable"
                faultName="..."/>
```

5.5.4.4.2.7 Fault handling

Fault handlers defined within a scope handle faults which are received from within that scope. A fault could arise due to

- the service requesting a service from an external service provider who returns a fault message, or
- the service making use of a lower level service which signals a fault.

5.5.4.4.2.8 What is a fault occurs within a process, but is not handled within the process

If a fault is raised at some level of granularity, but is not handled, even at the outer process level, then the termination handler for the process is called. The default fault handler

- first executes all compensation handlers for the current scope, and then
- rethrows the fault to the outer (calling) scope.

5.5.4.4.2.9 Defining fault handlers

Fault handlers are defined within the scope to which they apply. The scope could be

- a particular invocation,
- some scope of the business process as defined within an encasing `<scope>` tag, or

- the enclosing process itself.

The general syntax is as follows:

```
<scope>
    ...
    <faultHandlers>
       <catch faultName="someNameSpacePrefix:SomeFaultName"
           faultElement="someNameSpacePrefix:SomeElementName">
        <!--
            any activities to be executed for the fault handling
        -->
        </catch>
        <catch faultName="someNameSpacePrefix:SomeFaultName">
         <!--
                any activities to be executed for the fault handling
          -->
        </catch>
        <catchAll>
            <!--
                any activities to be executed for the fault handling
            -->
        </catchAll>
    </faultHandlers>
</scope>
```

5.5.4.4.2.10 Ignoring faults

At times one catches a fault which does not manifest itself in a fault for the current business process. In such cases one can use an `<empty>` activity to ignore the fault.

5.5.4.4.2.11 Inline fault handling

Fault hadnling can be specific for a particular invocation. If this is the case, one can use inline fault handling. This is done via an embedded `catch` clause within the `invoke` element.

5.5.4.4.2.12 Compensation handlers

Since an enterprise services bus often executes Long Running Transactions (LRTs), the default undo operations cannot use transaction rollback as this would require locking the resources which participate in the LRT for the duration of the LRT. The primary roll back mechanism for ESB hosted business processes is thus usually that of using compensating transactions which effectively undo a transaction through a series of inverse activities.

Compensation handlers are meant to execute a compensating transaction which effectively neutralizes the effects of a transaction witout rolling back the transaction. This is defined in compensation handlers which perform the inverse activities to undo the activities which have been done within the scope in which the compensation handler is defined.

The compensation handler is thus defined as a sequence of activities:

```
<scope>
  <compensationHandler>
    <!-- activities -->
  </compensationHandler>
</scope>
```

5.5.4.4.2.13 Propagating faults via rethrow

At times one defines a fault handler for a particular scope in order to perform certain clean up activities, without actually handling the fault. In such cases one would like to `<rethrow>` the fault for further fault handling at higher levels of the services hierarchy.

5.5.4.4.3 Structured activities

Structured activities specify a higher level flow logic across basic activities.

5.5.4.4.3.1 Sequences of activities

A sequence of activities is used to specify a sequential process across worklow steps which are done completed one after the other.

The typical structure of a sequential process looks as follows:

```
<process>

    <partnerLinks>
      <partnerLink .../>
      ...
    </partnerLinks>

   <variables>
      <variable ...>
      ...
   <variable>

   <sequence>
      <receive .../>

      <assign .../>

      <invoke .../>  <!-- synchronous request -->

      <invoke .../>  <!-- asynchronous request with subsequent waiting for a response -->
      <receive .../>

      <assign .../>

      <invoke .../>

      <reply .../>
   </sequence>

</process>
```

5.5.4.4.3.2 Flow control using switch

To support optional flow within a busines process, BPEl supports the concept of a switch construct.

The general syntax for a switch is

```
<switch>
    <case condition="(boolean expression)">
        <!-- some activity -->
    </case>
    ...
    <case condition="(boolean expression)">
        <!-- some activity -->
    </case>
</switch>
```

In order to support typical functionality required for condition specification, BPEL extends the XPath vocabulary to support a `getVariableData` function. The syntaxt for the `getVariableData` function is

```
bpws:getVariableData('variable-name',
            'part-name',
            'location-path')
```

with the latter to parameters being optional and the location-path query being specified in the query language selected for this business process specification.

For example, we could switch on the total value of an assessment via

```
<switch>
      <case condition="bpws:getVariableData(
                         'assessment', 'valuation','/valuation/ass:totalValue') @gt; 1e6">
        <-- activity to be performed for assesments above a million Rand -->
      </case>
      <case condition="bpws:getVariableData(
                         'assessment', 'valuation','/valuation/ass:totalValue') @gt;  ↩
                             100000">
        <-- activity to be performed for assesments above a 100000 Rand -->
      </case>
      <otherwise>
        <-- activity to be performed for assesments below R100000 -->
      </otherwise>
```

5.5.4.4.3.3 Concurrent activities via flow

The `flow` element is used to specify that certain spects of a business process could be executed concurrently. This is used for aspects of the business process which do not have dependencies on each other.

The typical structure of a sequential process looks as follows:

```
<process>

    <partnerLinks>
      <partnerLink .../>
      ...
    </partnerLinks>

   <variables>
      <variable ...>
      ...
   <variable>

   <sequence>
      <receive .../>

      <assign .../>

      <invoke .../>

      <flow>  <!-- The 2 sequences in this flow are executed concurrently -->
        <sequence>

          <invoke .../>
          <receive .../>

        </sequence>

        <sequence>

           <invoke>
          <invoke .../>
          <receive .../>

        </sequence>
      <flow>

      <assign .../>

      <invoke .../>

      <reply .../>
   </sequence>

</process>
```

5.5.5 WS-BPEL and manual work flow steps

WS-BPEL itself does not currently have explicit support for manual (human) work flow steps - it merely interacts with service providers and consumers. Most WS-BPEL processes will be executed in a container which is plugged into an ESB, however, and the ESB may offer any number of binding components (protocol adaptors) through which a service request may be routed to a human, and the response returned to the BPEL process.

5.5.6 The Benefits of WS-BPEL

When it comes to implementing a business process, BPEL enables a process-centric development style, where the services that participate in a single business process do not have to be split up (with potentially complex and repetitive code to gain access to the *state* of the business process) across different deployed artifacts. It enables a more natural implementation path because it introduces a physical, centralised artifact for the business process. From the client's point of view, however, services do remain stateless, de-coupled and autonomous.

Another major benefit is that it serves as a public standard for implementing complex integration scenarios, affording portability across containers and ESBs, whether they adhere to common low-level standards such as JBI or SCA, or not. Historically, such complex integration scenarios have depended on the availability of specific rules and EIP (Enterprise Integration Patterns) engines, but few if any of them have the centralised, simplified elegance that BPEL provides in a portable way.

5.5.7 The Drawbacks of WS-BPEL

WS-BPEL appears to be slow in supporting newer XML standards, with it currently only mandating support for XSLT and XPath 1.0, which severely limit the processing ability when managing low-level information, such as performing date/time processing. This means that a WS-BPEL implementation may require more low-level, de-coupled service providers than what may have been necessary, even if such service providers exist only for the purpose of the WS-BPEL process (i.e. not intended for re-use).

The implementation complexity around declaring message properties, partner links, as well as the extreme degree of abstraction (such as where a WSDL partnerlink declares an abstract role, the BPEL binds (abstractly) to certain roles, and then the WS-BPEL container / ESB often needs another mapping to concretely bind the abstract roles to the deployed services) make it non-trivial to develop BPEL processes, even using sophisticated tools. The promise of easy implementation by non-technical persons are tus theoretically possible, but today's editors have not realised this vision yet.

Part III

Designing an SOA-based solution

The importance of design in SOA

The challenges that the technologically-diverse SOA environment pose to the developer are significant. This, coupled with the current lack of efficient and intuitive developer tools, means that it is absolutely important for any SOA development to be performed based on strong design principles.

Chapter 6

URDAD for technology neutral, services-centric business process design

6.1 Introduction

URDAD is a technology neutral business process design methodology. URDAD does not target SOA based systems exclusively. Instead it is meant to be a technology-neutral design methodology to be embedded within a model-driven approach. However, URDAD is ideally suited for designing SOA based systems since it is

- services centric,
- contract centric,
- directly supports compose-ability of services, and
- provides the outputs required for a services-oriented design.

6.2 Designing for a Services-Oriented Architecture

When designing for a deployment within a services oriented architecture one requires as outputs of the design process

- the services contracts including
  - the services across levels of granularity,
  - the services offered with their pre- and post-conditions,
  - the messages exchanged with their associated data structures,
  - the quality requirements for the services,
  - the business processes through which these services are realized.

URDAD as a technology-neutral, services- and contract-centric business process design methodology provides these outputs.

6.3 URDAD - the methodology

6.3.1 Introduction

URDAD provides a use case driven algorithmic analysis and design methodology generating a technology neutral business process design. One thus starts with a concrete user service for which the business process is to be designed.

The methodology enforces those design activities which have been identified as drivers for the desired design attributes and the resultant business benefits.

One of the core aspects of URDAD is the consistent management of level of granularity. Each level of granularity has both, an analysis and a design phase. This ensures that one does not have to identify all requirements across all levels of granularity up-front.

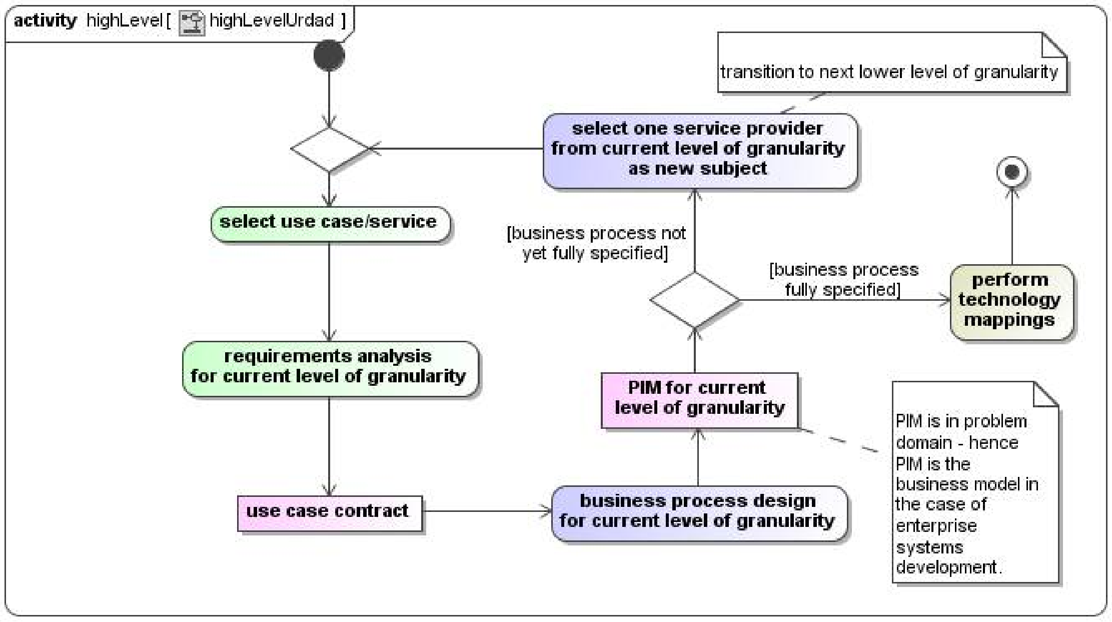

activity highLevel[highLevelUrdad]
transition to next lower level of granularity
select one service provider from current level of granularity as new subject
select use case/service
[business process not yet fully specified]
[business process fully specified]
perform technology mappings
requirements analysis for current level of granularity
PIM for current level of granularity
PIM is in problem domain - hence PIM is the business model in the case of enterprise systems development.
use case contract
business process design for current level of granularity

Figure 6.1: High-level view of the URDAD methodology

The methodology starts with the initial analysis phase followed by a design phase for the current level of granularity. This will project out the high level service providers required to realize the use case. One of these is selected as the subject for the next lower level of granularity. One then repeats the process for each of the lower level services (use cases) required from that services provider. One thus performs the requirements analysis followed by the business process design for each of these lower level services.

Figure **??** shows the URDAD analysis and design methodology in more detail.

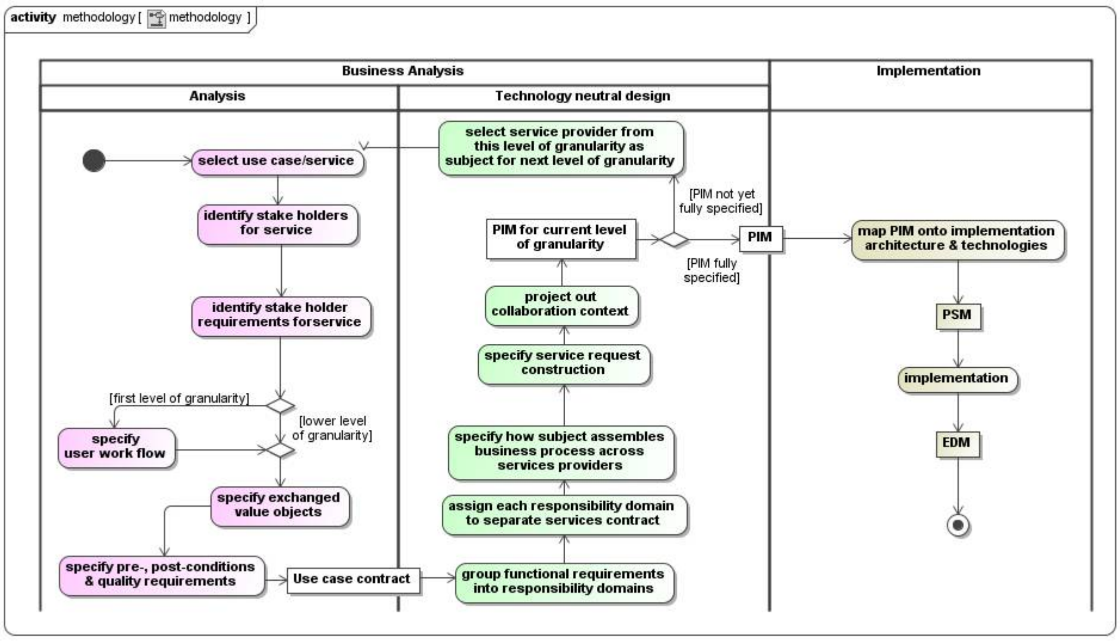

activity methodology [methodology]
Business Analysis
Implementation
Analysis
Technology neutral design
select use case/service
identify stake holders for service
identify stake holder requirements forservice
[first level of granularity]
[lower level of granularity]
specify user work flow
specify exchanged value objects
specify pre-, post-conditions & quality requirements
Use case contract
group functional requirements into responsibility domains
assign each responsibility domain to separate services contract
specify how subject assembles business process across services providers
specify service request construction
project out collaboration context
PIM for current level of granularity
[PIM not yet fully specified]
[PIM fully specified]
select service provider from this level of granularity as subject for next level of granularity
PIM
map PIM onto implementation architecture & technologies
PSM
implementation
EDM

Figure 6.2: More detailed outline of the URDAD methodology

We use the example of processing an insurance claim to illustrate the algorithm. This example is taken through two levels of granularity in order to illustrate the incremental refinement of the technology neutral business process design.

6.3.2 The analysis phase

6.3.2.1 Introduction

The analysis phase aims to elicit, verify and document the stake holder requirements. As one takes the business process design through lower levels of granularity, one revisits the analysis phase to elicit the lower level requirements around the individual workflow steps.

Often the high level analysis around the core business process and the lower level analysis around a functional requirement from a specific responsibility domain is done by different business analysts who focus on different business areas.

For example, a business analyst from the claims department would analyze the requirements and design the high level business process for for processing a claim. The lower level requirements and business processes around how, for example, the claim is to be paid out or how the claim is to be valued could be performed by business analysts from the finance and valuations department of the organization.

6.3.2.2 Functional requirements

During the analysis phase one first identifies all those stake holders who have an interest in the use case. Stake holders are those objects who place requirements around a use case. Only once one has identified all the stake holders in a use case can one hope to elicit all functional requirements for that use case. Maintaining the linkage of any functional requirement to the stake holder who requires it facilitates full traceability of any business or system activity back to the stake holder requirements they realize and ultimately to the stake holder itself.

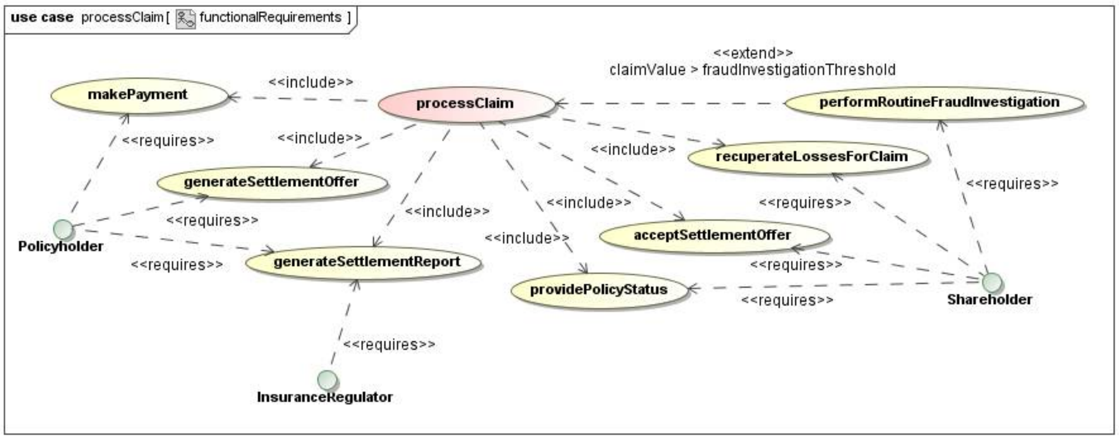


Figure 6.3: Functional requirements for the process claim use case.

Figure **??** shows the high level functional requirements for a process claim use case. Only the first level functional requirements should be included at this level of granularity. The detailed lower level functional requirements around the higher level ones are specified at lower levels of granularity.

For example, the functional requirement of *generating a settlement offer* may include lower level functional requirements like that of determining the value of the claim items and that of assessing to what extend the p0olicy covers the claim.

Note
Often the detailed requirements around the different domains of responsibility are obtained from different role players; i.e. while certain domains of business may be able to provide information around the higher level business process, the details concerning lower level responsibilities are often determined from domain experts in the appropriate domains of responsibility.

6.3.2.2.1 Pre-conditions, post-conditions and quality requirements

In contract-driven development a services contract is specified by the

- service signature with inputs and outputs,
- pre-conditions,
- post-conditions, and
- quality requirements.

The pre-conditions are those conditions under which the service may be refused without breaking the contract.

The post-conditions are those conditions which must hold once the service has been provided. They apply to the success scenarios of the use case. Each functional requirement directly maps onto a post-condition.

Finally, there may be quality requirements which are specific to this use case. Quality requirements are non-functional requirements referring to the realizable quality of service [BCK_2003_SAIP]. They refer to aspects like scaleability, reliability, performance, integrability, ... and are the core drivers behind architecture and infrastructure. While the pre- and post-conditions are part of the functional requirements which are realized through design, the quality requirements are used to assess whether the target architecture for the use case can indeed host the use case or whether architectural adjustments need to be made in order to realize the required quality requirements.

6.3.2.2.1.1 Bibliography

[BCK_2003_SAIP] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, second edition, April 2003, Addison-Wesley Professional.

6.3.2.3 User work flow

The required user work flow is documented via interaction diagrams showing only the messages exchanged between the subject responsible for realizing the use case and the actors.

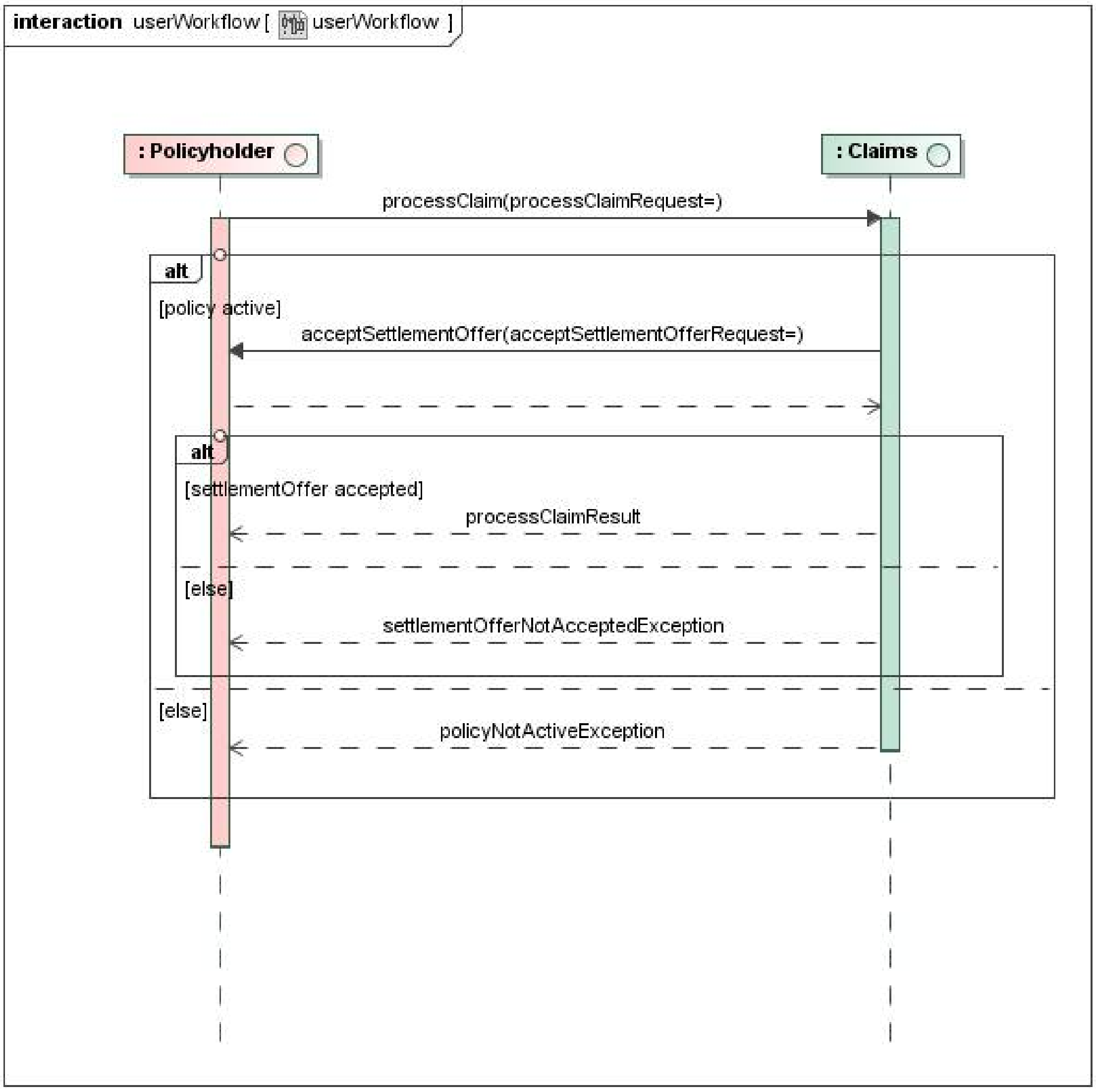

interaction userWorkflow [userWorkflow]
: Policyholder
: Claims
processClaim(processClaimRequest=)
alt
[policy active]
acceptSettlementOffer(acceptSettlementOfferRequest=)
alt
[settlementOffer accepted]
processClaimResult
[else]
settlementOfferNotAcceptedException
[else]
policyNotActiveException

Figure 6.4: The user work flow for a success scenario of the use case.

For example, Figure **??**, shows the interactions of the subject with the actors for a particular scenario and specifies the value objects exchanged between them.

6.3.2.4 The services contract

In contract-driven development a services contract is specified by the

- service signature with inputs and outputs,
- class diagrams specifying the data structure requirements for the inputs and outputs,

- pre-conditions,
- post-conditions, and
- quality requirements.

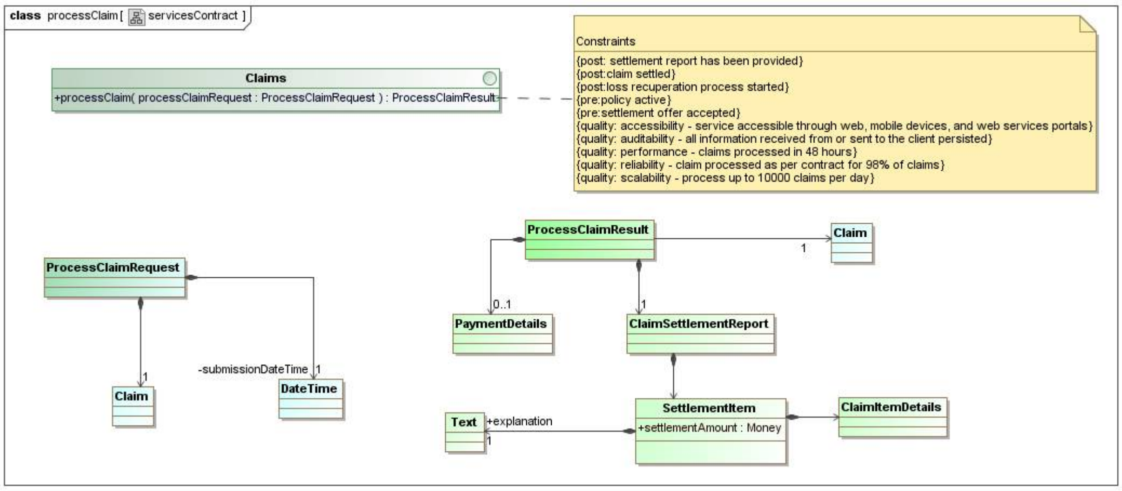

class processClaim [servicesContract]
Claims
+processClaim(processClaimRequest : ProcessClaimRequest) : ProcessClaimResult
Constraints
{post: settlement report has been provided}
{post:claim settled}
{post:loss recuperation process started}
{pre:policy active}
{pre:settlement offer accepted}
{quality: accessibility - service accessible through web, mobile devices, and web services portals}
{quality: auditability - all information received from or sent to the client persisted}
{quality: performance - claims processed in 48 hours}
{quality: reliability - claim processed as per contract for 98% of claims}
{quality: scalability - process up to 10000 claims per day}
ProcessClaimResult
Claim
1
ProcessClaimRequest
0..1
1
PaymentDetails
ClaimSettlementReport
1
-submissionDateTime
1
Claim
DateTime
Text
+explanation
1
SettlementItem
+settlementAmount : Money
ClaimItemDetails

Figure 6.5: The services contract for the process claim service.

6.3.3 The design phase

6.3.3.1 Defining the use case contract

During an URDAD design phase one identifies the responsibilities for the current level of granularity, assigns them to services contracts and specifies the business process the role players realizing the contract need to execute. One then projects out the collaboration context, i.w. the static structure supporting the collaboration which realizes the use case.

The output of the design phase is the technology neutral business process design for that level of granularity. This will include

- the service providers required for the current level of granularity,
- the business process for the current level of granularity,
- the collaboration context which resembles, in a technology neutral way, that subset of the static structure required to support the business process for the current level of granularity, and
- class diagrams for any object exchanged between the role players of the current level of granularity.

6.3.3.2 Responsibility identification and allocation

During the first step of an URDAD design phase one groups functional requirements into responsibility domains and assigns each responsibility domain to a separate services contract. Note that the technology neutral design assigns responsibilities not to concrete implementation classes, but instead to service provider contracts. These contracts can be realized by implementations in different technologies.

In the context of a model driven development process, the choice of a concrete service provider or the technology within which a service provider is o be realized is made during the implementation mapping phase. A services contract can be realized by a system, a system component, an organizational component (e.g. a business unit) or an external service provider to whom the organization has outsourced the responsibility covered by the services contract.

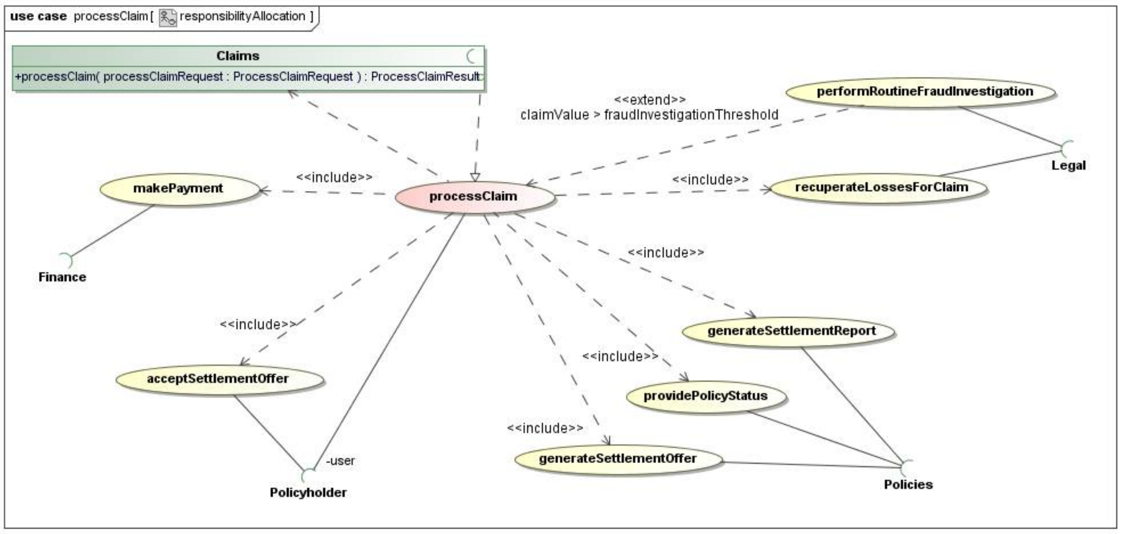

use case processClaim [responsibilityAllocation]
Claims
+processClaim(processClaimRequest : ProcessClaimRequest) : ProcessClaimResult
<<extend>>
claimValue > fraudInvestigationThreshold
performRoutineFraudInvestigation
Legal
<<include>>
makePayment
processClaim
recuperateLossesForClaim
Finance
generateSettlementReport
acceptSettlementOffer
providePolicyStatus
generateSettlementOffer
-user
Policyholder
Policies

Figure 6.6: Responsibility identification and allocation for the process claim use case

URDAD requires that one adds the responsibility for managing the work flow and assigns the responsibility to a separate services contract. This decouples the service providers from one another, localizes the business process information for the current level of granularity and removes any business process information from the service providers themselves. They are simply there to provide reusable services around a responsibility domain without knowledge of the business processes for which these services are required.

Figure **??** shows the responsibility identification and allocation for the process claim use case.

6.3.3.3 Business process specification

The services contracts are first introduced abstractly without specifying the services which service providers realizing the services contract need to provide. Instead one next designs the business process for the current level of granularity, showing how these abstract service providers need to collaborate in order to realize the use case. The business process design feeds the services required for the business process into the services contracts for the service providers required for the business process.

6.3.3.3.1 The activity diagram specifying the business process for the processClaim service

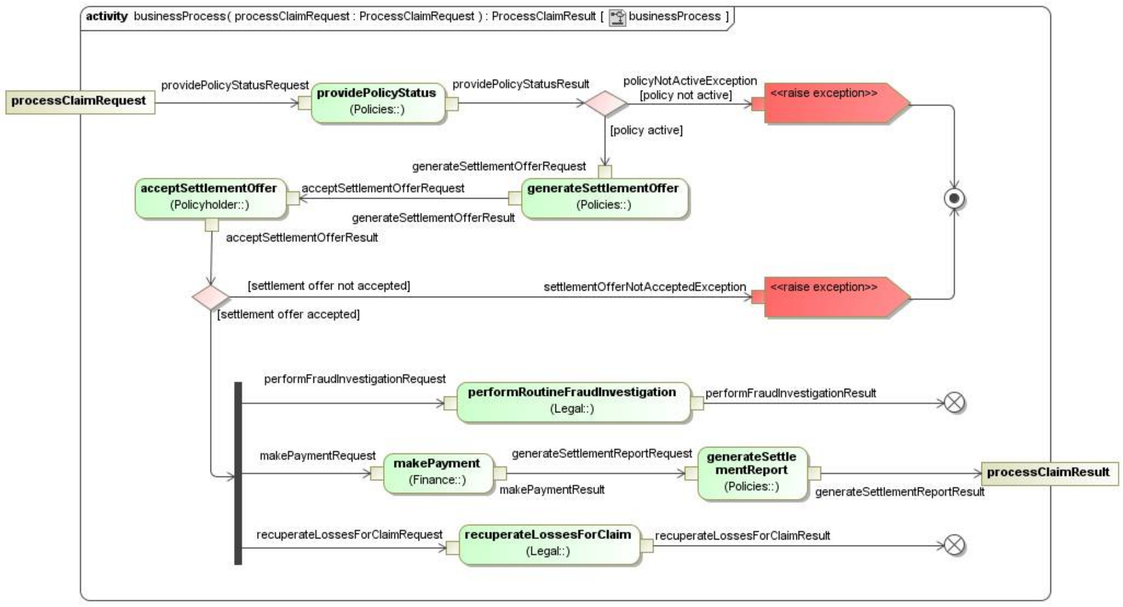

activity businessProcess(processClaimRequest : ProcessClaimRequest) : ProcessClaimResult [businessProcess]
processClaimRequest
providePolicyStatusRequest
providePolicyStatus (Policies::)
providePolicyStatusResult
policyNotActiveException [policy not active]
<<raise exception>>
[policy active]
generateSettlementOfferRequest
acceptSettlementOffer (Policyholder::)
acceptSettlementOfferRequest
generateSettlementOffer (Policies::)
generateSettlementOfferResult
acceptSettlementOfferResult
[settlement offer not accepted]
settlementOfferNotAcceptedException
<<raise exception>>
[settlement offer accepted]
performFraudInvestigationRequest
performRoutineFraudInvestigation (Legal::)
performFraudInvestigationResult
makePaymentRequest
makePayment (Finance::)
generateSettlementReportRequest
generateSettlementReport (Policies::)
makePaymentResult
processClaimResult
generateSettlementReportResult
recuperateLossesForClaimRequest
recuperateLossesForClaim (Legal::)
recuperateLossesForClaimResult

Figure 6.7: Activity diagram showing how the controller assembles the business process across services sourced from service providers.

6.3.3.4 Projecting out the collaboration context

The collaboration context shows the service providers required, at a specific level of granularity, to realize the use case, the services they need to provide for this use case and the message paths we require in order for the service providers to be able to collaborate to realize the use case.

Figure **??** shows the collaboration context for the process claim use case. Note that the dynamics (i.e. the business process specification) will already have fed in the services required for the business process into the contracts for the individual service providers.

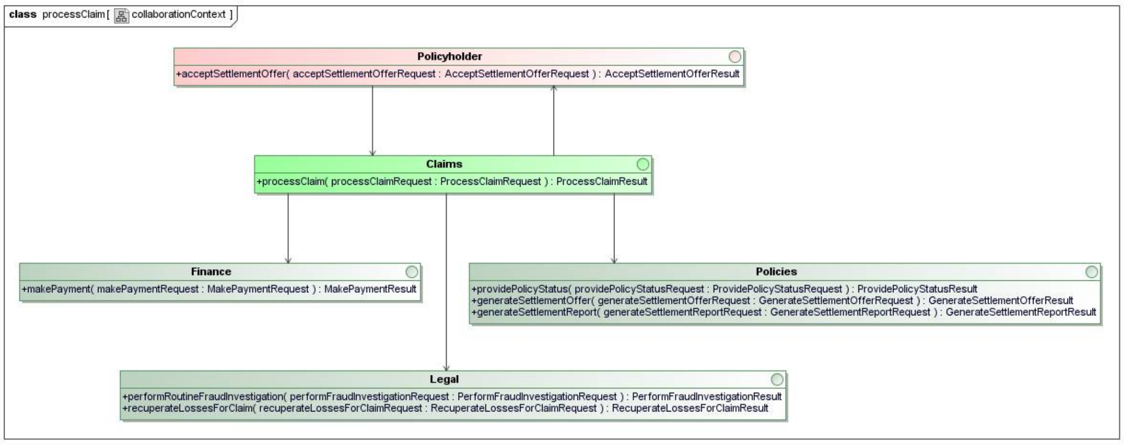

class processClaim [collaborationContext]
Policyholder
+acceptSettlementOffer(acceptSettlementOfferRequest : AcceptSettlementOfferRequest) : AcceptSettlementOfferResult
Claims
+processClaim(processClaimRequest : ProcessClaimRequest) : ProcessClaimResult
Finance
+makePayment(makePaymentRequest : MakePaymentRequest) : MakePaymentResult
Policies
+providePolicyStatus(providePolicyStatusRequest : ProvidePolicyStatusRequest) : ProvidePolicyStatusResult
+generateSettlementOffer(generateSettlementOfferRequest : GenerateSettlementOfferRequest) : GenerateSettlementOfferResult
+generateSettlementReport(generateSettlementReportRequest : GenerateSettlementReportRequest) : GenerateSettlementReportResult
Legal
+performRoutineFraudInvestigation(performFraudInvestigationRequest : PerformFraudInvestigationRequest) : PerformFraudInvestigationResult
+recuperateLossesForClaim(recuperateLossesForClaimRequest : RecuperateLossesForClaimRequest) : RecuperateLossesForClaimResult

Figure 6.8: The collaboration context for the process claim use case

6.3.4 Transition to next level of granularity

Having completed one analysis/design cycle, one needs to ask oneself whether the business process for the use case has been fully specified or not. If not, one may need to go to lower levels of granularity for some or all of the service providers from the current level of granularity.

Note

Often the lower level granularity design is done by different business analysts who understand that domain of responsibility (e.g. from a different department of the organization) or by the business analysts of other organizations to whom the realization of the services contract is outsourced.

In order to execute the transition to the next lower level of granularity, one selects one of the service providers as the new context. The services from the current level of granularity become the lower level use cases. After all, a use case is defined as a service of value[**?**]. One then selects a particular service or use case and repeats the lower level analysis and design process.

Figure **??** shows an example of stake holders around the lower level use case of providing a settlement offer together with their functional requirements around that use case.

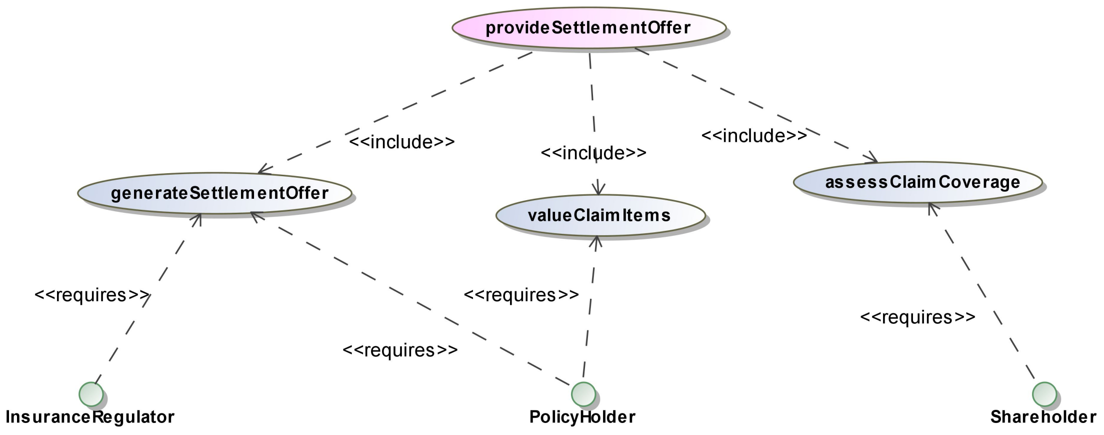


Figure 6.9: Functional requirements for the provide settlement offer service

The lower level design phase is executed in the same way as one was done for the higher level of granularity. It start with the grouping of functional requirements into responsibility domains and the allocation of each responsibility domain to a separate services contract. Figure **??** Figure \ref{fig:assessCoverageResponsibilityAllocation} shows an example of identifying and allocating the lower level responsibilities around assessing the policy coverage.

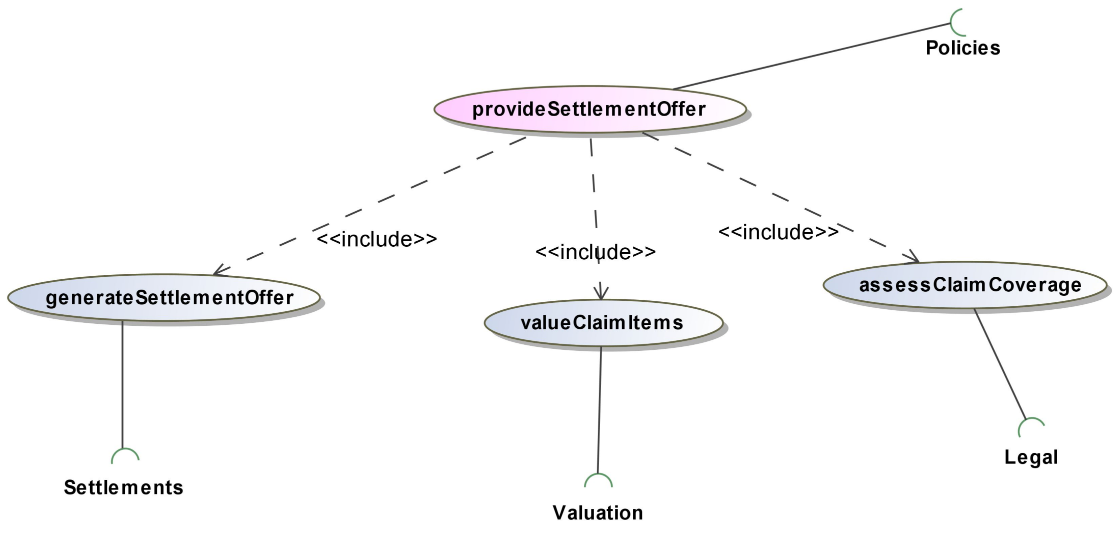


Figure 6.10: Responsibility allocation for the provide settlement offer service

6.3.4.1 Facilitating navigation across levels of granularity

In URDAD a service from one level of granularity is mapped onto a use case at the next lower level of granularity. In order to be able to conveniently navigate across levels of granularity, one needs to maintain the link between the the service and its corresponding use case. This can be done in various UML tools by adding a link with an appropriate stereotype.

6.3.4.2 Bibliography

[Frankel_2003_MDAA] James Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified Modeling Language Reference Manual*, ISBN 978-0471319207, 2004, second edition, Addison-Wesley Professional.

6.4 URDAD views

URDAD defines six standard views (diagrams) which can be used to construct and present an URDAD model. Three views are part of the analysis for the service/use case whilst the other 3 views are part of the The views include

- the *services contract view* specifying the stake holder requirements,
- the *user work flow view* specifying the required interaction between the user and the service provider
- the *functional requirements view* specifying the functionality (lower level services) required to address the stakeholder requirements,
- the *responsibility allocation view* specifying the services contracts to which the functional requirements are assigned to,
- the *process specification view* specifying the (business) process through which the service is realized, and
- the *collaboration context view* specifying the role players (contracts) collaborating in the process realizing the service and the

6.5 How are the design activities realizing the desired design attributes embedded in URDAD?

The single responsibility principle is directly enforced by grouping functional requirements into responsibility domains and requiring the each responsibility domain is assigned to a separate contract.

The levels of granularity are fixed by including only those contracts to which the responsibilities for a particular level of granularity have been assigned. Furthermore, the level of granularity is further fixed be requiring that the lowest level service requests at a particular level of granularity are those which come from the controller for that level of granularity.

The locking into services contracts is enforced by directly assigning responsibility domains to contracts and specifying the work flow across these contracts. The URDAD design process then generates the contract details and requires the specification of pre-and post-conditions as well as quality requirements.

URDAD directly enforces the introduction of a work flow controller for each responsibility domain and each level of granularity, resulting in the localization of the business process information and decoupling of the service providers used in the business process.

The relationship between the layers of granularity are documented through an explicit transition across the layers of granularity, facilitating bidirectional traceability.

Finally, the minimal conceptual (technology neutral) structure supporting the collaboration is projected out from the dynamics of the business process realizing the use case.

6.6 Evaluating an URDAD based design

In order to assess an URDAD based design one will

- validate that each functional requirement is indeed addressed by the business process,
- assess the grouping of functional requirements into responsibility domains in order to verify that there are no overlaps between responsibility domains and that each responsibility domain does indeed comprise a single responsibility,
- verify that the process at any level of granularity is intuitive and simple,
- verify that the service providers are represented by services contracts (UML interfaces) and not by implementation or technology specific classes,
- verify that each services contract has been fully specified including the functional and non-functional requirements,
- verify that the structure of all exchanged value objects is defined using class diagrams.

6.7 Implementation mappings

The implementation mappings to a particular SOA technology suite (such as a JBI-based EJB) would be quite technology-specific. Thus, while the technology neutral business process design is usually done by business analysts, the implementation mappings are usually done by the technical team, with guidance from the architect.

Often a business process is realised across a combination of manual (human) work flow steps, services provided by external service providers and automated processing steps executed within systems. The services contracts coming out of the technology neutral business process design can be used as a basis for the service provider contracts which are either realised by external service providers or by business units hosted within the organisation. The implementation mapping of such work flow steps may require training certain staff members to execute them.

Often, however, the technology mapping may result in mapping the technology neutral design onto

- a realisation using current systems, with perhaps some additional development,

- buying technology components and customising them, or
- developing an entire system hosting the various services from scratch.

MDA (Model-Driven Architecture) tools aim to automate this process fully in the near future.

6.7.1 Notes on mapping onto a Service-Oriented architecture (SOA)

6.7.1.1 Services Contracts

Services contracts are mapped to WSDL contracts which are typically abstract, i.e. without protocol binding definitions. The structure of the exchanged value objects are mapped to XML Schema definitions.

6.7.1.2 Workflow controllers

The service which plays the role of workflow controller at a certain level of granularity typically needs to invoke other, lower-level services. For this, we typically map it to a technology which could be used for service orchestration, such as *WS-BPEL* or a *Java-based* service.

The design for a workflow controller often involves dynamically routing requests to lower-level service providers based on certain criteria (message contents, context, environmental state). This implies a content-based router, often implemented in a rules or workflow technology such as Apache Camel or Drools.

Note
WS-BPEL has practical shortcomings in terms of data manipulation / querying when dealing with object-oriented XML structures, which render it practical only for relatively simple, course-grained and high-level activities, forcing the developer to outsource logic to a greater number of low-level services (implemented in, say, Java or XSLT) than what may have been otherwise necessary).

6.7.1.3 Service Adaptors

An URDAD-based design often includes, or implies, adaptors between services. These adaptors often primarily involve message transformation, to which a technology such as XSLT is very suited. If an adaptor contains a large amount of logic, or requires multiple helper services, it could be implemented as a special form of workflow controller.

6.7.1.4 Low-level business services

Individual services that perform atomic tasks (such as computation, database persistence, etc) are often implemented in a technology which has access to the necessary support technologies, such as Java (and often in a container such as EJB).

6.8 Summary

The set of accepted design principles which are seen as required characteristics of a good design can be supported by a set of design activities through which these design principles are realized. URDAD defines an algorithmic design process which incorporates these design activities. It generates a technology neutral business process design in the form of services contracts for each level of granularity together with the business process for that level of granularity. URDAD can be embedded within a model driven development process where the technology neutral business process design is ultimately mapped onto one's choice of implementation architecture and technologies.

Chapter 7

Managing your JBI-based infrastructure to realise quality of service

The JBI Specification (version 1.0) addresses almost no quality-of-service aspects of hosting an ESB. The core focus on the first version of the specification appears to have been almost purely functional. As such, different ESB vendors compete on the front of offering certain qualities of service, although (in practice) most ESB implementations seem to offer a similar feature set in this regard. We can thus make general statements and propose general strategies for achieving certain service qualities:

7.1 Reliability

7.1.1 High Availability

Although JBI describes a logically centralised infrastructure (i.e. a "single" normalised message router), most ESBs can be clustered across multiple physical hosts. This is usually achieved in the context of configuring the underlying (typically JMS-based) messaging system used by the Normalized Message Router (for example, Apache ActiveMQ used by the Servicemix ESB).

To achieve high reliability, a cluster of ESBs are configured such that a single instance is ‘active’, with the other instances being in ‘stand-by’ mode. Should the active instance go down, one of the stand-by instances takes over as the active instance.

This scenario achieves high availability without incurring any of the complexity of true clustering. Apache Servicemix, for example, calls this ‘passive clustering’.

7.1.2 Reliable Message Delivery

The ESB’s underlying messaging system usually allows vendor-specific configuration to trade off messaging performance against reliability. In its most reliable form, message delivery involves the transactional storage of messages in a database during message routing, ensuring that the ESB can survive system failures by continuing in a consistent state when the system is brought back up again.

7.2 Scalability

7.2.1 Clustering

To achieve a high degree of scalability, multiple instances of one’s ESB can be run across multiple network hosts. They are all configured to actively join a cluster group, after which they can all see each other’s components and services. In this scenario, messages can be seamlessly routed between ESBs as if all endpoints were defined in a single ESB.

In most current implementations, though the ESB infrastructure is clustered (duplicated across hosts), the service units themselves are not. Different service units are usually deployed to different hosts.

7.3 Auditability

One of the biggest advantages of mediated message exchange via the NMR is that sophisticated auditability (such as logging messages for future inspection) can be enabled. This can usually be done through one of two mechanisms:

- Configuring logging handlers for the normalised message router, such as plugging in a specific Log4J or `java.util.logging` handler into the specific ESB's messaging logger.
- Configuring a global message listener, usually via an ESB-specific API

Unfortunately, JBI 1.0 does not directly specify an interception API, so it is possible that one's auditability solution may be somewhat locked into a particular ESB, or at least its logging system (e.g. Log4J).

Recall, however, that JBI *does* specify standard administration and monitoring via JMX. Most of the MBeans support real-time statistics and monitoring, effectively providing another avenue through which to perform certain auditing tasks.

7.4 Security

7.4.1 Authentication

Authentication is usually performed in Binding components, when external clients make requests. User credentials are then typically stored in the incoming message (as JBI message headers) as the message is routed to different services.

7.4.2 Authorisation

JBI 1.0 does not specify any standard mechanism to perform authorisation (such as role-based authorisation) for services. Most ESB implementations, however, allow one to specify the authorisation requirements on a per-service level. If an authenticated service request is routed to such a service endpoint, it may be refused (by means of a fault message) if the client is not allowed to invoke the service.

7.4.3 Confidentiality

The Normalised Message Router itself inherently represents a trusted environment. As such, there is usually no mechanism to specify confidential message exchange between service endpoints on the ESB. Binding components, however, may employ several protocol-level confidentiality features (such as SSL / HTTPS) to ensure the confidential exchange of messages between external client(s) and the binding component.

Chapter 8

Synopsis and closing

8.1 Alternatives to the typical SOA infrastructure

The traditional definition of SOA usually implies an infrastructure that incorporates mediated message exchange via an Enterprise Services Bus (ESB). However, a number of other standards and approaches exist which would allow the construction of a solution which could easily also be called a Services-Oriented solution (but without the ESB):

8.1.1 SCA (Service Components Architecture)

Service Component Architecture (SCA) provides a programming model for building applications and systems based on a Service Oriented Architecture. It is based on the idea that business function is provided as a series of *services*, which are assembled together to create solutions that serve a particular business need. These *composite applications* can contain both new services created specifically for the application and also business function from existing systems and applications, reused as part of the composition. SCA provides a model both for the composition of services and for the creation of service components, including the reuse of existing application function within SCA compositions.

8.1.1.1 Who supports SCA ?

SCA is managed by the OASIS standards body, and has been put in place by collaboration of various industry partners, such as IBM, BEA, IONA Technologies, SAP, Siemens, and Oracle.

Note
A number of these partners are/were also involved in the competing JBI standard for SOA-based systems.

8.1.1.2 Primary SCA Concepts

SCA is composed of a number of standards (called ‘component parts’) which specify or govern some aspect of SCA-based SOA development:

- **Assembly Model** The SCA Assembly Model consists of a series of artifacts which define the configuration of an SCA system in terms of service components which implement and/or use services and the connections which describe how they are linked together. The assembly is defined in terms of a set of SCA composites, which define components and reference the implementation code that provide business function and which also describe services and references and the wires that link them.

- **Policy Framework** The capture and expression of non-functional requirements such as security is an important aspect of service definition, and has impact on SCA throughout the lifecycle of components and compositions. SCA provides the Policy Framework to support specification of constraints, capabilities and Quality of Service (QoS) expectations, from component design through to concrete deployment. This specification describes the framework and its usage.
- **Bindings** SCA Bindings apply to services and references. Bindings allow services to be provided, and references to be satisfied, via particular transports. There is a binding available for each different access method.

SCA itself is defined to be independent of different technologies and programming languages, with defined bindings for various programming languages and component models, such as EJB, WS-BPEL, Spring, PHP and Python.

8.1.2 Jini

A Jini system is a distributed system based on the idea of federating groups of users and the resources required by those users. The overall goal is to turn the network into a flexible, easily administered tool with which resources can be found by human and computational clients. Resources can be implemented as either hardware devices, software programs, or a combination of the two. The focus of the system is to make the network a more dynamic entity that better reflects the dynamic nature of the workgroup by enabling the ability to add and delete services flexibly.

A Jini system consists of the following parts:

- A set of components that provides an infrastructure for federating services in a distributed system
- A programming model that supports and encourages the production of reliable, distributed services
- Services that can be made part of a federated Jini system and that offer functionality to any other member of the federation

Although these pieces are separable and distinct, they are interrelated, which can blur the distinction in practice. The components that make up the Jini technology infrastructure make use of the Jini technology programming model; services that reside within the infrastructure also use that model; and the programming model is well supported by components in the infrastructure.

The end goals of the system span a number of different audiences; these goals include the following:

- Enabling users to share services and resources over a network
- Providing users easy access to resources anywhere on the network while allowing the network location of the user to change
- Simplifying the task of building, maintaining, and altering a network of devices, software, and users

The Jini technology infrastructure provides mechanisms for devices, services, and users to join and detach from a network of services. Joining and leaving a Jini system are easy and natural, often automatic, occurrences. Jini systems are far more dynamic than is currently possible in networked groups where configuring a network is a centralized function done by hand, or where central infrastructure is relied upon. Jini is highly de-centralised.

8.1.2.1 Primary Jini concepts

8.1.2.1.1 Services

The most important concept within the Jini architecture is that of a *service*. A service is an entity that can be used by a person, a program, or another service. A service may be a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user.

Members of a Jini system federate to share access to services. A Jini system should not be thought of as sets of clients and servers, users and programs, or even programs and files. Instead, a Jini system consists of services that can be collected together for the performance of a particular task. Services may make use of other services, and a client of one service may itself be a service with clients of its own. The dynamic nature of a Jini system enables services to be added or withdrawn from a federation at any time according to demand, need, or the changing requirements of the workgroup using the system.

Jini systems provide mechanisms for service construction, lookup, communication, and use in a distributed system. Examples of services include: devices such as printers, displays, or disks; software such as applications or utilities; information such as databases and files; and users of the system.

Services in a Jini system communicate with each other according to Services contracts (called the Service protocol), which is a set of interfaces written in the Java programming language. The set of such protocols is open ended. The base Jini system defines a small number of such contracts that define critical service interactions.

8.1.2.1.2 Lookup Service

Services are found (dynamically, by other services, instead of being statically wired together) and resolved by a lookup service. The lookup service is the central bootstrapping mechanism for the system and provides the major point of contact between the system and users of the system. In precise terms, a lookup service maps interfaces indicating the functionality provided by a service to sets of objects that implement (from the clien'ts perspective) the service. In addition, descriptive entries associated with a service allow more fine-grained selection of services based on properties understandable to people.

The lookup service is completely de-centralised, discovered via multicast networking, and may be federated / duplicated across the network.

8.1.2.1.3 Infrastructure services

Jini standardises certain infrastructure services, such as

- A distributed events model (for event-driven architecture, or observable processes)
- A leasing model (time-based, exclusive or non-exclusive access to resources)
- A distributed Transactions model
- An implementation of the Space-based ('Blackboard') architecture model (`JavaSpaces`) which allow for location-transparent, shared memory spaces that enable grid engines, computing farms, and highly de-centralised and self-innovative business processes.

8.2 Criticism of current JBI technologies

8.2.1 Trading off technical complexity against vendor lock-in

For practical purposes, most JBI-based development require the developer to either

- use low-level, vendor-neutral tool sets such as Apache Maven, together with a high degree of technical skill to configure composite integration assemblies by hand - and often brittle and error-prone task, or
- lock into vendor tools that allow for graphical and wizard-driven construction of composite assemblies, relying on the tool to generate the numerous configuration artifacts which likely lock the developer into a particular ESB as well.

8.2.2 Centralisation

Though several particular implementations of JBI-based ESBs can be federated or clustered across multiple machines, the ESB is still an inherently centralised infrastructure. If the Normalised Message Router ceases to function correctly, services will typically no longer be able to communicate with one another.

Certain approaches like Jini are inherently decentralised, with services discovering one another and forming spontaneous peer-to-peer networks. Together with an infrastructure like Rio for service provisioning and management, it represents an altogether more flexible and potentially more reliable infrastructure, in the spirit of 'cloud computing'.

8.2.3 Java-only

The JBI API caters only for ESB implementations written in the Java programming language. This is usually not a practical problem, as JBI service units accepts service units written in a very wide variety of technologies, and Java itself is cross-platform.

8.3 Anticipated future developments

8.3.1 Standardisation of SOA infrastructure

It is expected that the current ‘competition’ between several standards (JBI, SCA, OSGi, perhaps even Jini/Rio) will settle. Instead of one of these technologies claiming the prize of ultimate standard, an approach such as the Eclipse Swordfish platform may in fact combine many of them into one runtime environment, with each technology finding its niche in contributing to the whole.

8.3.2 Maturation of tools

Most SOA tools still require large amounts of work to be usable by developers with weaker technical skills. Current toolkits seem to either *expose too many*, or *hide too many* of the technicalities/control of a JBI project.

8.3.3 Entrenchment of practical Model-Driven development

Working in many ways against the goal of maturing the SOA tools, the process of Model-Driven Development will ultimately involve generating all code artifacts (static and dynamic aspects) from a Plaform-Independent model (PIM), such as an URDAD-compliant UML model.

Once the necessary transformation are in place to generate all SOA artifacts from a PIM, maintainability / ease of use will no longer be a strong factor in the selection of the implementation technology for an SOA service.

8.3.4 Adoption of semantic technologies

As SOA continues to show that any attempts at standardising the functional aspects of a service (service interface, exchanged value objects) between different clients are futile, more and more adaptors will be built to absorb different client requirements. Using semantic technologies (RDF/OWL) to represent information - first internally, stored in the database to flexibly cater for all client needs, and later externally as exchanged value objects - will free clients and services from the burden of representing all information in rigid class structures. We will instead by able to rely on meaningful semantic descriptions of information.

RDF, which represents information as graphs of statements, much more accurately enables business context-specific information to be added to continuously-growing knowledge repositories, and for use cases to evolve in unexpected ways without having to require that technical experts put representations for all possible scenarios in place in the form of the rigid little boxes that we call classes and interfaces.

Chapter 9

Index