Service-oriented design and development methodology

Michael P. Papazoglou*
and Willem-Jan van den Heuvel

INFOLAB, Department of Information Systems and Management
Tilburg University
P.O. Box 90153, Tilburg 5000 LE, The Netherlands
E-mail: mikep@uvt.nl
E-mail: wjheuvel@uvt.nl
*Corresponding author

Abstract: Service Oriented Architectures (SOA) are rapidly emerging as the premier integration and architectural approach in contemporary, complex, heterogeneous computing environments. SOA is not simply about deploying software: it also requires that organisations evaluate their business models, come up with service-oriented analysis and design techniques, deployment and support plans, and carefully evaluate partner/customer/supplier relationships. Since SOA is based on open standards and is frequently realised using Web Services (WS), developing meaningful WS and business process specifications is an important requirement for SOA applications that leverage WS. Designers and developers cannot be expected to oversee a complex service-oriented development project without relying on a sound design and development methodology. This paper provides an overview of the methods and techniques used in service-oriented design and development. The aim of this paper is to examine a service development methodology from the point of view of both service producers and requesters and review the range of elements in this methodology that are available to them.

Keywords: service oriented computing; Service Oriented Architecture (SOA); business processes; Web Services (WS); design and development methodologies.

Reference to this paper should be made as follows: Papazoglou, M.P. and van den Heuvel, W-J. (2006) 'Service-oriented design and development methodology', *Int. J. Web Engineering and Technology*, Vol. 2, No. 4, pp.412–442.

Biographical notes: Michael P. Papazoglou is a Professor of Computer Science and Director of the INFOLAB and CRISM at the University of Tilburg in The Netherlands. His research interests include distributed systems, service-oriented computing and WS, enterprise application integration, and e-business technologies and applications. He received a PhD in Computer Systems Engineering from the University of Edinburgh.

Willem-Jan van den Heuvel is an Associate Professor of Information Systems at the University of Tilburg in The Netherlands. His research interests include service-oriented computing, alignment of new enterprise systems with legacy systems, and system evolution. He received a PhD in Computer Science from the University of Tilburg.

1 Introduction

Service Oriented Architectures (SOAs) provide a set of guidelines, principles and techniques by which business processes, information and enterprise assets can be effectively (re)organised and (re)deployed to support and enable strategic plans and productivity levels that are required by competitive business environments (Papazoglou and Georgakapoulos, 2003). In this way, new processes and alliances need to be routinely mapped to services that can be used, modified, built or syndicated.

Many enterprises in their early use of SOA suppose that they can port existing components to act as WS just by creating wrappers and leaving the underlying component untouched. Since component methodologies focus on the interface, many developers assume that these methodologies apply equally well to service-oriented architectures. As a consequence, implementing a thin SOAP/WSDL/UDDI layer on top of existing applications or components that realise the WS is by now widely practised by the software industry. Yet this is in no way sufficient to construct commercial-strength enterprise applications. Unless the nature of the component makes it suitable for use as a WS, and most are not, it takes serious thought and redesign effort to properly deliver components' functionality through a WS. While relatively simple WS may be effectively built that way, a service-based development methodology is of critical importance to specify, construct, refine and customise highly volatile business processes from internally and externally available WS. More importantly, older software development paradigms for object-oriented and component-based development cannot be blindly applied to SOA and WS.

In this paper, we concentrate on the workings of a services design and development methodology that provides sufficient principles and guidelines to specify, construct, refine and customise highly volatile business processes choreographed from a set of internal and external WS. The objective of the service-oriented design and development methodology is to achieve service integration as well as service interoperability. The objective of the service-oriented design and development methodology is to achieve service integration as well as service interoperability.

In Section 2, we describe the general characteristics of a services design and development methodology, while in Section 3, we introduce the phases in the methodology and in Section 4, we describe useful design and development principles for WS. In Section 5, we describe the phases of this methodology in some detail while Section 6 presents our conclusions and outlook.

2 Characteristics of service development life cycle methodology

A Web Services Life Cycle Development (Papazoglou, 2006) methodology should focus on analysing, designing and producing an SOA in such a way that it aligns with business process interactions between trading partners in order to accomplish a common business goal, *e.g.*, requisition and payment of a product, and stated functional and nonfunctional business requirements, *e.g.*, performance, security, scalability and so forth.

Service-oriented design and development incorporates a broad range of capabilities, technologies, tools and skill sets, which include (Arsanjani, 2004; Brown *et al.*, 2005):

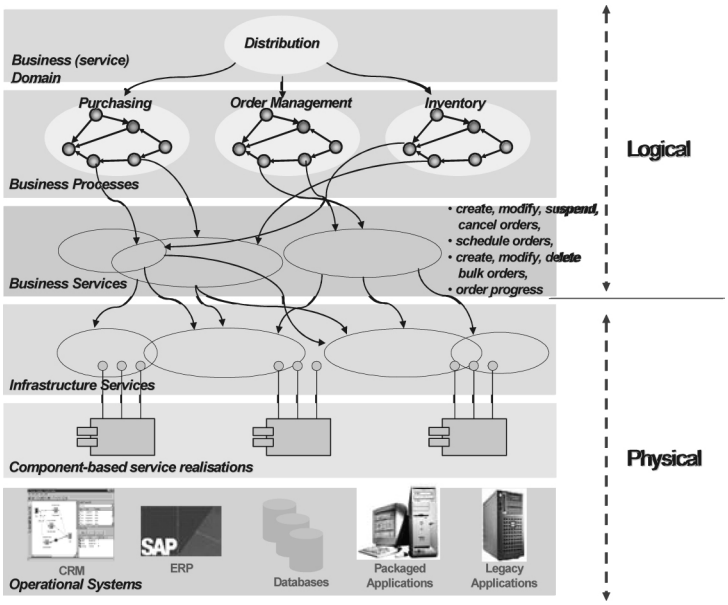
- managing the entire services life cycle – including identifying, designing, developing, deploying, finding, applying, evolving, and maintaining services
- establishing a platform and programming model, which includes connecting, deploying and managing services within a specific runtime platform
- adopting best practices and tools for architecting service-oriented solutions in repeatable, predictable ways that deal with changing business needs
- delivering high quality workable service-oriented solutions that respect Quality of Service (QoS) requirements. These solutions may be implemented on best practices, such as tried and tested methods for implementing security, ensuring performance, compliance with standards for interoperability and designing for change.

Service design and development is about identifying the right services, organising them in a manageable hierarchy of composite services (smaller grained often supporting larger grained), choreographing them together for supporting a business process. A business service or process can be composed of finer-grained (atomic) services that need to be supported by infrastructure services and management services, such as those providing technical utility like logging, security or authentication, and those that manage resources.

Classifying related business processes that exhibit common functional characteristics and objectives can raise the level of abstraction in an SOA. In this way, business process conglomerations can be created and organised under a service domain. Classifying business processes into logical service domains simplifies an SOA by reducing the number of business processes and services that need to be addressed. A *service domain*, also referred to as *business domain*, is a functional domain comprising a set of current and future business processes that share common capabilities and functionality and can collaborate with each other to accomplish a higher-level business objective, *e.g.*, loans, insurance, banking, finance, manufacturing and human resources. In this way a business can be portioned into a set of disjointed domains. Such domains can be leveraged by multiple architectural reasons such as load balancing, access control, and vertical or horizontal partitioning of business logic.

Figure 1 shows that a service domain such as distribution is subdivided into higher-level business processes such as purchasing, order management and inventory. In this figure, the order management business process, which we shall use as a running example throughout the paper, performs order volume analysis, margin analysis, sales forecasting and demand forecasting across any region, product or period. It can also provide summary and transaction detail data on order fulfilment and shipment according to item, sales representative, customer, warehouse, order type, payment term and period. Furthermore, it can track order quantities, payments, margins on past and upcoming shipments, and cancellations for each order. The order management process in Figure 1 is shown to provide business services for creating, modifying, suspending, cancelling and querying orders, and scheduling order activities. Business services can also create, modify and delete bulk orders and order activities, while customers could be informed of the progress of an order and its order activities. Business services in the order management process are used to create and track orders for a product, a service or a resource, and are used to capture the customer-selected service details. Information that is captured as part of an order may include customer account information, product offering and quality of service details, Service Level Agreement (SLA) details, access information, scheduling information, and so forth.

Figure 1 Web Services development life cycle hierarchy



Business services are supported by infrastructure, management and monitoring services. These services provide the infrastructure enabling the integration of services through the introduction of a reliable set of capabilities, such as intelligent routing, protocol mediation, and other transformation mechanisms, often considered part of the Enterprise Service Bus (Chappell, 2004; Papazoglou, 2006). This layer also provides the capabilities required for enabling the development, delivery, maintenance and provisioning of services, as well as capabilities that monitor, manage and maintain QoS, such as security, performance and availability. It also provides services that monitor the health of SOA applications, giving insights into the health of systems and networks, and into the status and behaviour patterns of applications, thus making them more suitable for mission-critical computing environments. Monitoring services implement all important standards implementations of WS-Management, and other relevant protocols and standards such as WS-Policy and WS-Agreement.

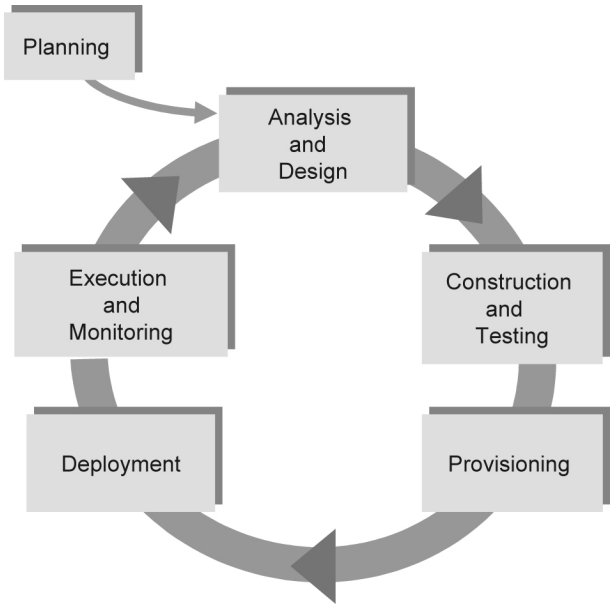
All service domains, business processes and services are automatically populated with financial and operational functions and data available from resources such as ERP, databases, CRM and other systems, which lie at the bottom of the service life cycle development hierarchy. However, component implementation is an issue that can seriously impact the quality of available services. Both services and their implementation components need to be designed with the appropriate level of granularity. The granularity of components should be the prime concern of the developer responsible for providing component implementations (see Section 4.3).

We may collectively think of the service domain, business processes and business services sections as constituting the *logical part* of the WS development life cycle, while we think of the infrastructure services, the component-based realisations, and the operational systems sections as constituting the *physical part* of the WS development life cycle.

3 Web services development life cycle methodology baseline

This section presents the elements of a service-oriented design and development methodology that is partly based on other successful related development models, such as the Rational Unified Process (RUP, 2001; Kruchten, 2004), Component-based Development (Herzum and Sims, 2000) and Business Process Modelling (Harmon, 2003), and concentrates on the levels of the WS development life cycle hierarchy depicted in Figure 2.

Figure 2 Phases of the service-oriented design and development methodology



A service-oriented design and development methodology is based on an iterative and incremental process that comprises one preparatory and eight distinct main phases that concentrate on business processes. These are planning, Analysis and Design (A&D), construction and testing, provisioning, deployment, execution and monitoring. These phases may be traversed iteratively (see Figure 2). This approach is one of continuous invention, discovery and implementation, with each iteration forcing the development team to drive the software development project's artefacts closer to completion in a predictable and repeatable manner. The approach considers multiple realisation scenarios for business processes and WS that take into account both technical and business concerns.

The planning phase constitutes a preparatory phase that serves to streamline and organise consequent phases in the methodology. During the planning phase, the project feasibility, goals, rules and procedures are set and requirements are gathered. The analysis phase is based on a thorough business case analysis that considers various alternatives for implementing business processes, *e.g.*, by wrapping enterprise systems or acquiring new services, while the design phase aims at identifying and specifying WS and business processes in a stepwise manner. Service construction and testing involves

coding WS and business processes using the specifications that were developed during the design phase. It also involves testing the coded services and processes for functional correctness and completeness as well as for interoperability. The service-provisioning phase then enforces the business model for service provisioning, which is chosen during the planning phase. This activity encompasses such issues as service metering, service rating and service billing. Once the provisioning model has been established, the WS may be deployed and advertised in a repository system. The final phase in the methodology deals with the execution and monitoring of WS. This phase includes the actual binding and runtime invocation of the deployed services, as well as managing and monitoring their life cycle.

The workings of this methodology will concern us for the rest of this paper and will be exemplified by means of the order management process that we introduced earlier.

4 Service-oriented design and development principles

A service-oriented design and development methodology focuses on business processes, which it considers to be reusable building blocks that are independent of applications and the computing platforms on which they run. This promotes the idea of viewing enterprise solutions as federations of services connected via well-specified contracts.

The course of designing a business process goes through the stages outlined earlier. However, in order to design useful and reliable business processes that are developed on the basis of existing or newly coded services, we need to apply sound service design principles that guarantee that services are self-contained and come equipped with clearly defined boundaries and service endpoints to allow for service composability. Two key principles serve as the foundation for service- and business-process design: service coupling and cohesion. We introduce in this section the concept of service granularity. Although service granularity is not a design principle it heavily impacts the characteristics and shape of services, as it is inextricably associated with service coupling and cohesion.

4.1 Service coupling

It is important that grouping of activities in business processes is as independent as possible from other such groupings in other processes. One way of measuring service design quality is coupling, or the degree of interdependence between two business processes. The objective is to minimise coupling, that is, to make (self-contained) business processes as independent as possible by not having any knowledge of or relying on any other business processes. Low coupling between business processes indicates a well-partitioned system that avoids problems of service redundancy and duplication.

Coupling can be achieved by reducing the number of connections between services in a business process, eliminating unnecessary relationships between them, and by reducing the number of necessary relationships – if possible. Coupling is a very broad concept, however, and for service design can be organised along the following dimensions:

- Representational coupling

  Business processes should not depend on specific representational or implementation details and the assumptions of one another, *e.g.*, business processes do not need to know the scripting language that was used to compose their underlying services. These concerns lead to the exploitation of interoperability and reusability for service design. Representational coupling is useful for supporting interchangeable/replaceable services and multiple service versions.

- Identity coupling

  Connection channels between services should be unaware of who is providing the service. It is not desirable to keep track of the targets (recipients) of service messages, especially when they are likely to change or when discovering the best service provider is not a trivial matter.

- Communication protocol coupling

  A sender of a message should rely only on those effects necessary to achieve effective communication. The number of messages exchanged between a sender and addressee in order to accomplish a certain goal should be minimal, given the applied communication model, *e.g.*, one-way, request/response and solicit/response. For example, the one-way style of communication, where a service endpoint receives a message without having to send an acknowledgement, places the lowest possible demands on the service performing the operation.

4.2 Service cohesion

Cohesion is the degree of the strength of functional relatedness of operations within a service. Service aggregators should create strong, highly cohesive business processes – business processes whose services and service operations are strongly and genuinely related to one another. A business process with highly related services and responsibilities, and which does not perform a tremendous amount of computational work, has high design cohesion. The guidelines by which to increase service cohesion are as follows:

- Functional service cohesion

  A functionally cohesive business process should perform one and only one problem-related task and contain only services necessary for that purpose. At the same time the operations in the services of the business process must also be highly related to one another, *i.e.*, highly cohesive. Consider services such as ‘get product price’, ‘check product availability’, and ‘check credit worthiness’, in an order management business process.

- Communicational service cohesion

  A communicationally cohesive business process is one whose activities and services use the same input and output messages. Communicationally cohesive business processes are cleanly decoupled from other processes as their activities are hardly related to activities in other processes.

- Logical service cohesion

  A logically cohesive business process is one whose services all contribute to tasks of the same general category by performing a set of independent but logically similar functions (alternatives) that are tied together by means of control flows. A typical example of this is mode of payment.

Like low coupling, high cohesion is a service-oriented design and development principle to keep in mind during all stages in the methodology. High cohesion increases the clarity and ease of comprehension of the design; simplifies maintenance and future enhancements; achieves service granularity at a fairly reasonable level; and often supports low coupling. Highly related functionality supports increased reuse potential, as a highly cohesive service module can be used for very specific purposes.

4.3 Service granularity

Service granularity refers to the scope of functionality exposed by a service. Services may exhibit different levels of granularity. An implementation component can be of various granularity levels. Fine-grained component (and service) implementations provide a small amount of business-process usefulness, such as basic data access. Larger granularities are compositions of smaller-grained components and possibly other artefacts, where the composition taken as a whole conforms to the enterprise component definition. The coarseness of the service operations to be exposed depends on business usage scenarios and requirements and should be at a relatively coarse level, reflecting the requirements of business processes.

A coarse-grained interface might be the complete processing for a given service, such as 'SubmitPurchaseOrder', where the message contains all of the business information needed to define a purchase order. A fine-grained interface might have separate operations for: 'CreateNewPurchaseOrder', 'SetShippingAddress', 'AddItem', and so forth. This example illustrates that fine-grained services might be services that provide basic data access or rudimentary operations. These services are of little value to business applications. Services of the most value are coarse-grained services that are appropriately structured to meet specific business needs. These coarse-grained services can be created from one or more existing systems by defining and exposing interfaces that meet business process requirements. WS interfaces may be invoked via messages, which may likewise be defined as coarse- or fine-grained entities.

Fine-grained messages result in increased network traffic and make handling errors more difficult. This significantly hinders cross-enterprise integration. However, internal use of WS may be beneficial in those cases where the internal network is faster and more stable. A higher number of fine-grained services and messages might therefore be acceptable for Enterprise Application Integration (EAI) applications.

From the perspective of service-oriented design and development, it is preferable to create higher-level, coarse-grained interfaces that implement a complete business process. This technique provides the client with access to a specific business service, rather than getting and setting specific data values and sending a large number of messages. Enterprises can use a single (discrete) service to accomplish a specific business task, such as billing or inventory control, or they may compose several services together to create a distributed e-business application such as customised ordering, customer

support, procurement and logistical support. These services are collaborative in nature and some of them may require transactional functionality. Enabling business to take place via limited message exchanges is the best way to design a WS interface for complex distributed applications that make use of SOAs.

5 Phases of the service-oriented design and development methodology

The phases of the service-oriented design and development methodology are described in some detail in this section. A detailed coverage of these phases is given in Papazoglou (2006).

5.1 The planning phase

The planning phase determines the feasibility, nature and scope of service solutions in the context of an enterprise. A strategic task for any organisation is to achieve a service technology 'fit' with its current environment. The key requirement in this phase is thus to understand the business environment and to make sure that all necessary controls are incorporated into the design of a service-oriented solution. Activities in this phase include analysing the business needs in measurable goals, reviewing the current technology landscape, conceptualising the requirements of the new environment and mapping those to new or available implementations. Planning also includes a financial analysis of the costs and benefits, including a budget and a software development plan, including tasks, deliverables and schedule.

A business environment is usually large and complex. Business experts at the service provider's side provide a categorisation and decomposition of the business environment into business areas based on the functions being served by sets of business processes. The discrete units of work done within an enterprise are organised as business processes within the business areas. Business processes are in reality higher-order services that are further decomposed into simpler services.

The planning phase is very similar to that of software development methodologies, including the RUP (RUP, 2001; Royce, 1998), and will be not discussed any further.

5.2 The analysis phase

Service-oriented analysis is a phase during which the requirements of a new application are investigated. This includes reviewing business goals and objectives that drive the development of business processes. Business analysts complete an 'as-is' process model to allow the various stakeholders to understand the portfolio of available services and business processes. The organisation designs, simulates, and analyses potential changes to the current application portfolio for potential Return-On-Investment (ROI) before it commits to any changes to business processes. This analysis results in the development of the 'to-be' process model that an SOA solution is intended to implement.

The analysis phase examines the existing services portfolio at the service provider's side to understand which policies and processes are already in place and which need to be introduced and implemented.

The analysis phase encourages a radical view of process (re)design and supports the reengineering of business processes. Its main objective is the reuse (or repurposing) of business process functionality in new composite applications. To achieve this objective the analysis phase comprises four main activities: process identification, process scoping, business gap analysis and process realisation.

5.2.1 Process identification

Understanding how a business process works and how component functionality differs or can get adjusted between applications is an important milestone when identifying suitable business process candidates. When designing an application, developers must first analyse application functionality and develop a logical model of what an enterprise does in terms of business processes and the services the business requires from them, *e.g.*, what are the shipping and billing addresses, what is the required delivery time, what is the delivery schedule and so on. Thus the objective of this step is to identify the services that need to be aggregated into a business process whose interface has a high viscosity.

The key factor is being able to recognise functionality that is essentially self-sufficient for the purposes of a business process. It is important when designing a business process to identify the functionality that should be included in it and the functionality that is best incorporated into another business process. Here, we can apply the design principles of coupling and cohesion to achieve this. For instance, an order management process has low communication protocol coupling with a material requirements process.

Process identification could start with comparing the abstract business process portfolio with standard process definitions, such as RosettaNet's 'Manage Purchase Order' (PIP3A4). This PIP encompasses four complementary process segments supporting the entire chain of activities from purchase order creation to tracking-and-tracing, each of which is further decomposed into multiple individual processes. A quick-scan of this PIP reveals that Segment 3A 'Quote and Order Entry' and Segment 3C 'Returns and Finance' may be combined into the process Order Management.

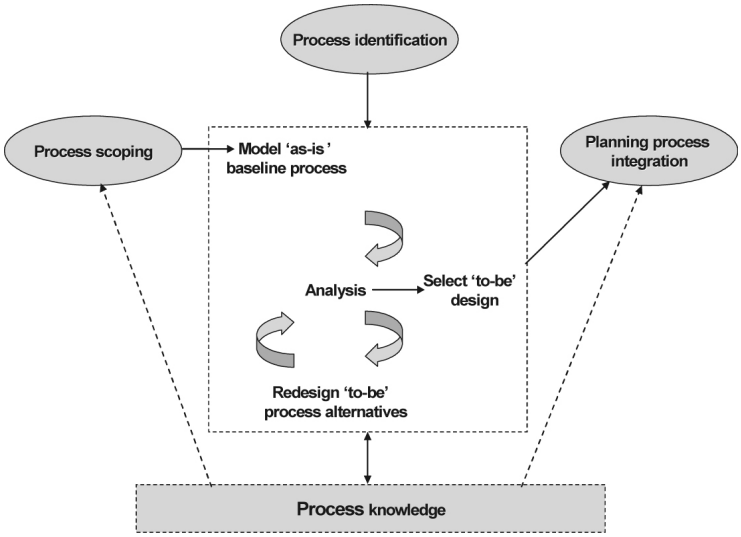
5.2.2 Process scoping

Defining the scope of business processes helps ensure that a process does not become monolithic, mimicking a complete application. Unbundling functionality into separate (sub)business processes will prevent business processes from becoming overly large, complex and difficult to maintain. For example, designing a business process that handles online purchasing would require the removal of packaging and shipping information and costs to different business processes. In this example, the three functions are mutually exclusive and should be implemented separately. The functionality included in these business processes is not only discrete and identifiable, but also loosely coupled to other parts of the application.

The *scope of a business process* is defined as an aggregation of aspects that includes where the process starts and ends, the typical customers (users) of the process, the inputs and outputs that the customers of the process expect to see, the external entities, *e.g.*, suppliers or logistics providers, that the process is expected to interface with, and the different types of events that start an instance of the process.

Figure 3 illustrates how business identification and scoping interact and result either in processes that can be reused and are candidates for design or in processes that need to be redesigned and reengineered.

Figure 3 Business process identification and scooping



Source: Adopted from El Sawy (2001)

5.2.3 Business gap analysis

Gap analysis is a technique that purposes a business process and WS realisation strategy by incrementally adding more implementation details to an abstract service/process interface. Gap analysis commences with comparing candidate service functionality with available software service implementations that may be assembled within the enclosures of a newly conceived business process. A gap analysis strategy may be developed in stages and results in a recommendation to do development work, or reuse or purchase WS. Several service realisation possibilities are discussed later in this section. For the moment let us assume that there may exist software components internal to an organisation that provide a good match. These may include service implementations previously developed by the enterprise, externally supplied service realisations available on a subscription or pay per use basis. Service realisations may be a blend of service and service-enabled implementations. In this way, portfolios of services possibly accessible on a global scale will complement and sometimes even entirely replace monolithic applications as the new fabric of business processes.

5.2.4 Process realisation analysis

Process realisation analysis is an approach that considers diverse business process realisation scenarios evaluated in terms of costs, risks, benefits and return of investment in accordance with business requirements and priorities. Service providers consider the following four realisation options (which may be mixed in various combinations) to develop new business processes (Brittenham, 2001):

1 Green-field development

   This step involves describing how a new interface for a WS will be created on the basis of the WS implementation. Green-field development assumes that first a service is implemented and subsequently the service interface is derived from the new WS implementation. During this step the programming languages and models that are appropriate for implementing the new WS are also determined.

2 Top-down development

   Using this realisation option a new service can be developed that conforms to an existing service interface. This type of service interface is usually part of an industry standard that can be developed by any number of service providers. Processes are usually deployed top-down from a business level process blueprint. The main benefit of the top-down service development is the consistency of the applications and integration mechanisms. It is also rather easy to evolve the service-oriented solution across the enterprise as the industry evolves. Main problems with this realisation option are the costs involved in development as well as the costs of achieving consensus on a high-level SOA architecture throughout the enterprise.

3 Bottom-up development

   Using this option, a new service interface is developed for an existing application. The existing application can be coded as a Java, C++ program, Enterprise Java Bean (EJB), *etc.*, or could be a back-end legacy application. This option usually involves creating a WS interface from the Application Programming Interface (API) of the application that implements the WS. Bottom-up development is well suited for an environment that includes several heterogeneous technologies and platforms or uses rapidly evolving technologies.

4 Meet-in-the-middle development

   This option is used when an already existing WS interface – for which an implementation already exists – is partially mapped onto a new service or process definition. This option involves service realisations that mix service and service-enabled implementations. This approach may thus involve creating a wrapper for the existing applications that need to be service-enabled and that are to be combined with the already existing WS interface. Meet-in-the-middle development realisation strategies offer a middle ground that attempts to take advantage of some of the benefits of the other approaches while extenuating some of the most notable problems and risks.

One of the issues with the top-down, bottom-up and meet-in-the-middle development options is that they are rather ambiguous regarding which business processes an enterprise should start from and how these can be combined to form business scenarios. To address this problem, service development solutions need to target specific focal points and common practices within the enterprise, such as those that are specified by its corresponding sector reference models. Reference models – a typical example of which is RosettaNet – address a common large proportion of the basic 'plumbing' for a specific sector, from a process, operational function, integration, and data point of view. Such a *verticalised development model* presents the ideal architecture for supporting service

development. This makes certain that the development team is aware of known best practices and standard processes so that they do not reinvent the wheel. For example, developers could use RosettaNet's standard processes PIP4B2 ('Notify of Shipment Receipt') and PIP4C1 ('Distribute Inventory Report') for applications in which suppliers develop replenishment plans for consignments. Product receipt and inventory information is notified using PIP4B2 and PIP4C1, respectively, between consignment warehouses and suppliers.

The various options for process realisation analysis emphasise the separation of specification from implementation, which allows WS to be realised in different ways, *e.g.*, top-down or meet-in-the-middle development. It then becomes important to plan effectively when deciding how to realise or provision services; we need to carefully examine the diversity of realisation alternatives and make the right choice. The service realisation strategy involves choosing from an increasing diversity of different options for services, in addition to service reuse, which may be mixed in various combinations. This includes reusing or repurposing already existing WS, business processes or business process logic; developing new WS or business processes logic from scratch; purchasing/leasing/paying per use for services; outsourcing service design and implementation regarding WS or (parts of) business processes; and using wrappers and/or adapters to revamp existing enterprise (COTS) components or existing (ERP/legacy) systems.

Process realisation results in a business architecture represented by business processes and the set of normalised business functions extracted from the analysis of these processes. During process realisation, analysis decisions are made whether to reuse a particular enterprise asset, *e.g.*, service or business process. To determine the quality of a specific asset, quality metrics are used that evaluate its flexibility, extensibility, maintainability, and level of cohesion and coupling. The process realisation analysis estimates existing and expected operational costs, integration costs, service and process customisation costs, service and process provisioning costs and architecture costs for each realisation scenario. Developers and project managers could use tools such as the IBM Rational Portfolio Manager to gain insight into the business benefits, costs and risks of the SOA services portfolio (Brown *et al.*, 2005). Architecture costs are associated with acquiring artefacts for realising the target architecture, including servers, specialised software, training and required network bandwidth.

5.3 The service design phase

Service analysis is logically succeeded by service design, during which conceptual processes and services are transformed into a set of related, platform-agnostic interfaces. Designing a service-oriented application requires developers to model and define well-documented interfaces for all major service components prior to constructing the services themselves. Service design is based on a twin-track development approach that provides two production lines: one to produce services (possibly out of preexisting components) and another to assemble (compose) services out of reusable service constellations. This calls for a business process model that forces developers to determine how services combine and interact jointly to produce higher-level services.

Service design, just like service analysis, has its own special characteristics and techniques, which we shall describe in this section. We shall first start by describing a broad set of service design concerns.

5.3.1 Service design concerns

A number of important concerns exist that influence design decisions and result in an efficient design of service interfaces, if taken seriously. These concerns bring into operation the design principles for service-enabled processes. Prime concerns include managing service granularity, designing for service reuse and designing for service composability.

Managing service and component granularity

Identifying the appropriate level of granularity for a service or its underlying component is a difficult undertaking, as granularity is very much application-context dependent. In general, there are several heuristics that can be used to identify the right level of granularity for services and implementation components. These include clearly identifiable business concepts, highly usable and reusable concepts, concepts that have a high degree of cohesion and low degree of coupling and must be functionally cohesive. Many vertical sectors, *e.g.*, automotive, travel industry and so on, have already started standardising business entities and processes by choosing their own levels of granularity.

Designing for service reusability

When designing services it is important to be able to design them for reuse so that they can perform a given function wherever this function is required within an enterprise. To design for service reuse one must make services more generic, abstracting away from differences in requirements between one situation and another, and attempting to use the generic service in multiple contexts where it is applicable. Designing a solution that is reusable requires keeping it as simple as possible. There are intuitive techniques that facilitate reuse that are related to design issues such as identification and granularity of services. These include looking for common behaviour that exists in more than one place in the system and trying to generalise behaviour so that it is reusable.

When designing a service-based application, it is possible to extract common behaviour and provide it by means of generic services so that multiple clients can use it directly. It is important, however, when designing enterprise services that business logic is kept common and consistent across the enterprise so that generalisation is not required. Nevertheless, there are cases wherein fine-tuning, specialisation or variation of business logic functionality is required. Consider, for instance, discounting practices that differ depending on the type of customer being handled. In those cases it is customary to produce a generalised solution with customisation points to allow for service variations.

Designing for service composability

In order to design useful and reliable services we need to apply sound service design principles that guarantee that services are self-contained, modular and support service composability. The design principles that underlie component reusability revolve around the two well-known software design guidelines – service coupling and cohesion.

5.3.2 Specifying services

A service specification is a set of three specification elements, all equally important. These are (Johnston, 2005):

1 Structural specification

   This focuses on defining the service types, messages, port types and operations.

2 Behavioural specification

   This entails understanding the effects and side effects of service operations and the semantics of input and output messages. If, for example, we consider an order management service, we might expect to see a service that lists 'place order', 'cancel order' and 'update order' as available operations. The behavioural specification for this ordering service might then describe how one cannot update or cancel an order you did not place, or that after an order has been cancelled it cannot be updated.

3 Policy specification

   This denotes policy assertions and constraints on the service. Policy assertions may cover security, manageability, *etc.*

During service design, service interfaces that were identified during the analysis phase are specified based on service coupling and cohesion criteria, as well as on the basis of the service design concerns that we examined in Section 6.1. In case that reference models are available, business processes and service interfaces can be derived on their basis. For the remainder of this section we consider a sample purchase order business process for supply chain service-oriented applications that are derived on the basis of RosettaNet's order management PIP cluster. By applying the dominant cohesion criterion, namely functional cohesion, this process may be decomposed into several subprocesses, such as 'Quote and Order Entry', 'Transportation and Distribution' and 'Returns and Finance', which conform to RosettaNet's segments 3A, 3B and 3C, respectively. The 'Quote and Order Entry' subprocess allows partners to exchange price and availability information, quotes, purchase orders and order status, and enables partners to send requested orders to other partners. The 'Transportation and Distribution' subprocess enables communication of shipping- and delivery-related information with the ability to make changes and handle exceptions and claims. Finally, the 'Returns and Finance' subprocess provides for issuance of billing, payment and reconciliation of debits, credits and invoices between partners, as well as supports product return and its financial impact. By applying functional cohesion again, the Quote and Order Entry subprocess may be decomposed into several services, such as 'Request Quote', 'Request Price and Availability', 'Request Purchase Order' and 'Query Order Status'. These four services conform to RosettaNet's PIPs 3A1, 3A2, 3A4 and 3A5, respectively.

Structural and behavioural service specification

In the following we will briefly examine the course of specifying an interface for a WS in WSDL (Chinnici *et al.*, 2004). WS interface specification comprises four steps: describing the service interface, specifying operation parameters, designating the messaging and transport protocol, and finally, fusing port types, bindings and actual location (a URI) of the WS. These steps themselves are rather trivial and already

described in-depth in literature, *e.g.*, in Alonso *et al.* (2004) a detailed approach for specifying services is outlined. The following guidelines and principles are relevant while developing a WSDL specification:

- The service interface should contain only port types (operations) that are logically related or functionally cohesive. For example, the service 'Request Purchase Order' captures the operations 'Purchase Order Request' and 'ReceiptAcknowledgement' as they are functionally cohesive.
- Messages within a particular port type should be tightly coupled by representational coupling and communication protocol coupling. For example, the operation 'Purchase Order Request' may have one input message (PurchaseOrderID) and one output message (PurchaseOrder) sharing the same communication protocol (*e.g.*, Simple Object Access Protocol (SOAP)) and representation (atomic XML Schema data types).
- Coupling between services should be minimised. For example, the services 'Request Purchase Order' and 'Query Order Status' are autonomous, having no interdependencies.

Specifying the service interface

A WSDL specification outlines operations, messages, types and protocol information. Figure 4 shows an abridged specification for a service interface and operation parameters for the 'Request Purchase Order' service. The WSDL example in Figure 4 illustrates that the WS defines two <portType> named 'CanReceive3A42_PortType' and 'CanSend3A42_PortType'. The <portType> 'CanReceive3A42_PortType' supports two <operation>, which are called 'PurchaseOrderRequest' and 'ReceiptAcknowledgement'.

Figure 4 WSDL excerpt for request purchase order

```
<wsdl:types>
      <xsd:complexType name = "PIP3A4PurchaseOrderRequest">
            <xsd:sequence>
                  <xsd:element ref = "PurchaseOrder"/>
                  <xsd:element ref = "fromRole"/>
                  <xsd:element ref = "toRole"/>
                  <xsd:element ref = "thisDocumentGenerationDateTime"/>
                  <xsd:element ref = "thisDocumentIdentifier"/>
                  <xsd:element ref = "GlobalDocumentFunctionCode"/>
            </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name = "PurchaseOrder">
            <xsd:sequence>
                  <xsd:element ref = "deliverTo" minOccurs = "0"/>
                  <xsd:element ref = "comment" minOccurs = "0"/>
                  <xsd:element ref = "packListRequirements" minOccurs = "0"/>
                  <xsd:element ref = "ProductLineItem" maxOccurs = "unbounded"/>
                  <xsd:element ref = "GlobalShipmentTermsCode"/>
                  <xsd:element ref = "RevisionNumber"/>
                  <xsd:element ref = "prePaymentCheckNumber" minOccurs = "0"/>
                  <xsd:element ref = "QuoteIdentifier" minOccurs = "0"/>
                  <xsd:element ref = "WireTransferIdentifier" minOccurs = "0"/>
                  <xsd:element ref = "AccountDescription" minOccurs = "0"/>
                  <xsd:element ref = "generalServicesAdministrationNumber"
                                     minOccurs = "0"/>
                  <xsd:element ref = "secondaryBuyerPurchaseOrderIdentifier"
                                     minOccurs = "0"/>
                  <xsd:element ref = "GlobalFinanceTermsCode"/>
                  <xsd:element ref = "PartnerDescription" maxOccurs = "unbounded"/>
                  <xsd:element ref = "secondaryBuyer" minOccurs = "0"/>
                  <xsd:element ref = "GlobalPurchaseOrderTypeCode"/>
            </xsd:sequence>
      </xsd:complexType>
</wsdl:types>
  … … … …
<message name="PurchaseOrderRequest">
      <part name="PO-body" type="tns:PIP3A4PurchaseOrderRequest"/>
</message>
```

Specifying operation parameters

After having defined the operations, designers need to specify the parameters they contain. A typical operation defines a sequence containing an input message followed by an output message. When defining operation parameters (messages), it is important to decide whether simple or complex types will be used. As WS become increasingly more complex, and message-based SOAP will be more appropriate, complex schemas need to be created. The <operation> 'PurchaseOrderRequest' in Figure 4 is shown to contain an output message 'PurchaseOrderRequest', which includes a single part named 'PO-body'. This part is shown in Figure 5 to be associated with the complex type 'PIP3A4PurchaseOrderRequest' that is further specified to the level of atomic (XSD) types in the compartment that is embraced with the <wsdl:types> tag. As Figure 5 indicates, well-factored WS often result in a straightforward <portType> element where the business complexity is moved into the business data declaration.

Figure 5 Specifying parameters for operations in Figure 4

```
   … … … …
<portType name="CanReceive3A42_PortType">
     <!-- name of operation is same as name of message -->
     <operation name="PurchaseOrderRequest">
            <output message="tns:PurchaseOrderRequest"/>
     </operation>
     <operation name="ReceiptAcknowledgement">
            <output message="tns:ReceiptAcknowledgment"/>
     </operation>
</portType>

<portType name="CanSend3A42_PortType">
     <!-- name of operation is same as name of message -->
     <operation name="PurchaseOrderConfirmation">
            <input message="tns:PurchaseOrderConfirmation"/>
     </operation>
     <operation name="ReceiptAcknowledgment">
            <input message="tns:ReceiptAcknowledgment"/>
     </operation>
     <operation name="Exception">
            <input message="tns:Exception"/>
     </operation>
</portType>
   … … … …
```

Several graphical WS development environments and toolkits exist today. These enable developers to rapidly create, view and edit services using WSDL and to manage issues such as correct syntax and validation, inspecting and testing WS and accelerating many common XML development tasks encountered when developing WS-enabled applications.

Service programming style

In addition to structural and behavioural service specification, the service programming style must also be specified during the service design phase. Determining the service programming style is largely a design issue, as different applications impose different programming-style requirements for WS. Consider for example an application that deals with purchase order requests, purchase order confirmations and delivery information. This application requires that request messages contain purchase orders in the form of XML documents while response messages contain purchase order receipts or delivery

information again in the form of XML documents. This type of application uses data-oriented WS. Moreover, there is no real urgency for a response message to follow a request immediately, if at all. In contrast to this, consider an application that provides businesses with up-to-the-instant credit standings. Before completing a business transaction, a business may need to check a potential customer's credit standing. In this scenario, a request would be sent to the credit check WS provider (*e.g.*, a bank) and processed, and a response indicating the potential customer's credit rating would be returned in real time. This type of WS relies on an RPC- or process-oriented programming style. In these types of applications the client invoking the WS needs an immediate response or may even require that the WS interact in a back-and-forth conversational way.

The use of document-based messaging promotes loose coupling. In general, there is no reason for the WS client to know the name of the remote methods. In document-based messaging, the WS subsystem receives the document, invokes the appropriate methods, and responds. Using document-based messaging results in loose coupling, since the message constructs the document, but does not indicate the steps to process that document. With document-based messaging, it is only the receiver that knows the steps to handle an incoming document. Thus, the receiver can add additional steps, delete steps, and so on, without impacting on the client.

Service policy concerns

The design of service-oriented solutions, like any other complex structure, requires early architectural decisions supported by well-understood design techniques, structural patterns and styles that go far beyond ensuring 'simple' functional correctness, and deal with nonfunctional service concerns. These patterns address common QoS issues that include performance requirements, information regarding service reliability, scalability and availability, transactional requirements, change management and notification, and so on. In general, nonfunctional service characteristics describe the broader context of a service, *e.g.*, what business function the service accomplishes, how it fits into a broader business process and characteristics of the hosting environment, such as whether the component provider ensures security and privacy, what kind of auditing, security and privacy policy is enforced by the component provider, what levels of quality of the component are available and so on.

Typical service-related nonfunctional concerns that are addressed by policies include security issues and authorisation concerns, and policy models. For example, a service provider could specify a policy stating that a given WS requires Kerberos tokens, digital signatures, and encryption. These can be used by clients use such policy information to determine whether they can use the particular service under consideration. In another example, an authentication model may require that a client authenticate itself by presenting its encoded credentials, or may require that XML signatures be generated for WS requests.

As a wide range of services is provided across a network, it is natural that services would benefit from the use of policy management models, which could determine the configuration of a network of differentiated services according to business rules or application-level policies. There are many reasons why enterprises might want to give different levels of service to different customers or why they might need different levels of priority for different business transaction models involving WS. Therefore, it is

only natural that such criteria constitute important elements of a service design methodology and be considered equally important to technical policies such as security or authentication.

5.3.3 Specifying business processes

Designers should be able to compose (or decompose) and relate to each other process models that are developed in different parts of the enterprise or by partners. They also should be able to incrementally refine processes and export process improvements achieved in one part of the business to other parts of the business, with adaptation as required. Industry best practices and patterns must also be taken into account when designing processes, and abstract process models can act as blueprints for subsequent concrete models.

Once business processes are extracted and their boundaries are clearly demarcated, they need to be described in the abstract. This step comprises three separate tasks, one deriving the process structure, one linking it to business roles, which reflect responsibilities of the trading partners, *e.g.*, a buyer, a seller and a shipper in the order management process, and one specifying nonfunctional characteristics of business processes. The first task is choosing the type of service composition. The choice is between orchestration versus choreography. If a choice for orchestration is made, three tasks follow to orchestrate a process. These are defined using the WS Business Process Execution Language (BPEL) (Andrews *et al.*, 2003). In the following we will place emphasis on orchestration, given that today there are several implementations for BPEL, while the WS-CDL (Choreography Description Language) is still being specified.

Describing the business process structure

The first step in the business process design is to specify the business structure and the functions of the business process. The business process structure refers to the logical flow or progression of the business process. A business process reveals how an individual process activity (<PortType>) is linked with another in order to achieve a business objective. To assemble a higher-level service (or process) by combining other WS, the service aggregator needs to select potential services that need to be composed depending on how these services and their operations fit within the enclosures of a business process and how they relate to one another. Subsequently, the service provider needs to connect the process interface to the interfaces of imported services and plug them together. Business processes can be scripted using BPEL.

The abstract description of a process encompasses the following tasks:

- Identify, group and describe the activities that together implement a business process

  The objective of this action is to identify the services that need to be combined in order to generate a business process and then describe the usage interface of the overall business process. The functions of a business process are expressed in terms of the activities or the services that need to be performed by a specific business process. For instance, the registration of a new customer is an activity in a sales order process. The structure of a business process describes how an individual process activity (<PortType>) is linked with one another. To assemble a higher-level service by combining other WS, the service designer needs to:

a select the services to compose by looking at how these services and their operations within a business process relate to one another

b connect the usage interface of the business process to the interfaces of imported services and plug them together.

- Describe activity dependencies, conditions or synchronisation

  A process definition can organise activities into varying structures such as hierarchical, conditional and activity dependency definitions. In hierarchical definition processes, activities have a hierarchical structure. For instance, the activity of sending an insurance policy for a shipped order can be divided into three sub-activities: compute the insurance premium, notify insurance, mail insurance premium to the customer. In process definitions that have a conditional activity structure, activities are performed only if certain conditions are met. For instance, it may be company policy to send a second billing notice to a trading partner when an invoice is more than two months overdue. Activity dependency definitions signify dependencies between activities and sub-activities in a process. In any process definition, sub-activities can execute only after their parent activity has commenced. This means that sub-activities are implicitly dependent on their parent activity. In other cases there might be an explicit dependency between activities: an activity may only be able to start when another specific activity has been completed. For instance, a shipment cannot be sent to a customer if the customer has not been sent an invoice.

- Describe the implementation of the business process

  Provide a BPEL definition that maps the operations and interfaces of imported services to those of another in order to create the usage interface of the business process (higher-level WS).

Figure 6 illustrates an abbreviated snippet of the BPEL specification for the order management process. The first step in the process flow is the initial buyer request. The listing shows that three activities are planned in parallel. An inventory service is contacted in order to check the inventory, and a credit service is contacted in order to receive a credit check for the customer. Finally, a billing service is contacted to bill the customer. Upon receiving the responses from the credit, inventory and billing services, the supplier would construct a message back to the buyer.

Figure 6 BPEL process flow for purchase order process

```
<sequence>
    <receive   partner="Manufacturer" portType="lns:PurchaseOrderPortType"
                    operation="Purchase"   variable="PO"
                    createInstance="yes" >
   </receive>
 <flow>
     <links>
          <link name="inventory-check"/>
          <link name="credit-check"/>
     </links>
    <!— Check inventory -->
    <invoke partner="inventoryChecker"
                portType="lns:InventoryPortType"
                operation="checkInventory"
                … … …
            <source linkName="inventory-check"/>
          …
    <invoke/>
    <!— Check  credit -->
    <invoke partner="creditChecker"
                portType="lns:CreditCheckPortType"
                operation="checkCredit"
                … … …
            <source linkName="credit-check"/>
         ….
    <invoke/>
    <!— Issue bill once inventory and credit checks are succesful -->
    <invoke partner="BillingService"
                portType="lns:BillingPortType"  operation="billClient"
                inputVariable="billRequest"  outputVariable="Invoice" >
                joinCondition="getLinkStatus("inventory-check") AND
                                        getLinkStatus("credit-check")" />
            <target linkName="inventory-check"/>
            <target linkName="credit-check"/>
         ….
    <invoke/>
 </flow>
   ...
 <reply partnerLink="Purchasing" portType="lns:purchaseOrderPT"
                          operation="Purchase" variable="Invoice"/>
</sequence>
```

Describing business roles

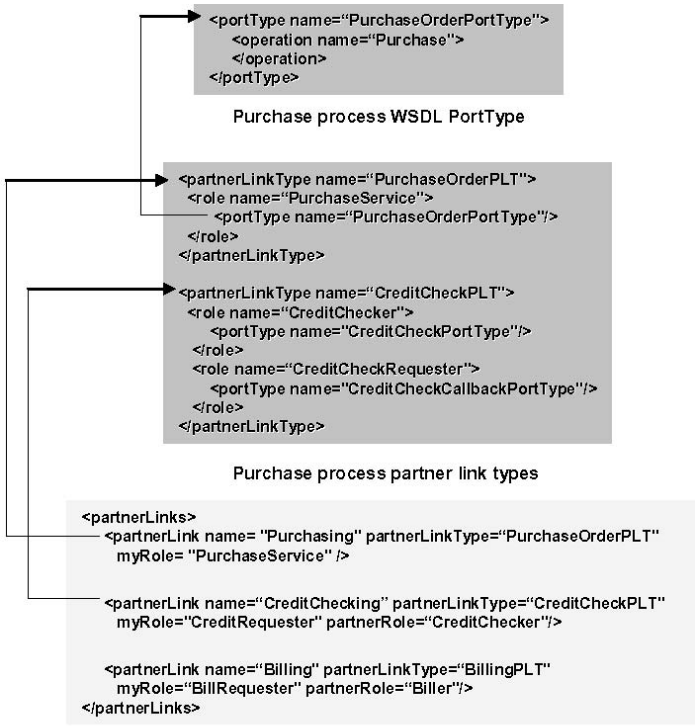
The second step during business process design is to identify responsibilities associated with business process activities and the roles that are responsible for performing them. Roles may thus invoke, receive and reply to business activities. Each service provider is expected to properly fulfil the business responsibility of implementing a business activity as one or more port types of a WS, which perform a specific role. The result of this phase actually constitutes the foundation for implementing business policies, notably role-based access control and security policies.

Figure 7 illustrates the different parties that interact within the business process in the course of processing a client's purchase order. This listing describes how the 'PurchaseOrder' service is interfaced with the port types of associated service specifications including a credit check and price calculation (billing) service (not shown in this figure) to create an order management process. Each <partnerLink> definition in this listing is characterised by a <partnerLinkType>.

Developers can use automated tools to assist them with designing and developing business processes. Toolsets such as IBM's WebSphere Business Modeller enables analysts to model, simulate and analyse complex business processes quickly and effectively. This toolset can be used to model 'as-is' and 'to-be' business processes, allocate resources and perform 'what-if' simulations to optimise and estimate business benefits (Brown *et al.*, 2005). These models can then be transformed into UML and Business Process Execution Language models to jumpstart integration activities. These descriptions can be used to orchestrate the constructed services as part of a business process workflow. Based on the overall business process model defined in WebSphere

Business Modeller and exported as a BPEL, developers bring together the overall workflow by wiring together service implementations. To achieve this they can use WebSphere Integration Developer to import, create, enact and manage business processes described in BPEL.

Figure 7 Defining roles in BPEL



Specialised graphical notations such as the Business Process Modelling Notation (BPMN) can also be used for modelling business processes. BPMN is an attempt at a standards-based business process modelling language that can unambiguously define business logic and information requirements to the extent that the resultant models are executable (White, 2004). BPMN is intended for allowing users to express the complex semantics of business processes. The net result is that business logic and information requirements are maintained in a model that is consistent with and reflects the requirements of business. BPMN is based on BPML's process execution meta-model and can produce directly fully executable BPEL processes.

Nonfunctional business process concerns

The process design subphase must also deal with nonfunctional process design concerns, including among other things performance, payment model, security model and transactional behaviour. In the following we will briefly mention typical business process-related, nonfunctional concerns.

Service Level Agreements provide a proven vehicle for not only capturing nonfunctional requirements but also monitoring and enforcing them. SLAs are special legal agreements that encapsulate multiple concerns, and symmetrically fuse the perspective of service provider and client. Besides mutual commitments regarding to-be-delivered services, *e.g.*, scalability and availability, the SLA should stipulate penalties, contingency plans for exceptional situations, and mechanisms for disaster recovery.

As an example, consider security policies targeted by an SLA. Such an SLA can be used to bundle security policies to protect multiparty collaborations. Knowing that a new business process adopts a WS security standard such as WS-Security (Atkinson *et al.*, 2002) is not enough information to enable successful composition. The client needs to know if the services in the business process actually require WS-Security, what kind of security tokens they are capable of processing, and which one they prefer. Moreover, the client must determine if the service should communicate using signed messages. If so, it must determine what token type must be used for the digital signatures. Finally, the client must decide on when to encrypt the messages, which algorithm to use, and how to exchange a shared key with the service. Trying to orchestrate services without understanding these technical details will inevitably lead to erroneous results. For example, the purchase order service in the order management process may indicate that it only accepts username tokens that are based on signed messages using X.509 certificate that is cryptographically endorsed by a third party.

5.4 The service construction phase

The construction phase of the life cycle methodology includes development of the WS implementation, the definition of the service interface description and the definition of the service implementation description (Brittenham, 2001). On the provider side the implementation of a WS can be provided by creating a new WS, or by transforming existing applications into WS, or by composing new WS from other (reusable) WS and applications. Unlike the previous phases that focus only on the provider, the service construction phase also considers service requesters. On the service client side, although the service requester progresses through similar life cycle elements as the service provider, different tasks are performed during each construction step. This phase may involve green-field code development; however, in most cases it will consist of modifying existing (J2EE or .NET) services or constructing wrappers on top of existing legacy applications. Thus, during the construction phase the process realisation scenario identified in Section 5.2.4 must be implemented.

5.5 The service test phase

Service testing is generally characterised as a validation exercise, ascertaining that requirements have been met and that the deliverables are at an acceptable level in accordance with existing standards during the analysis, design and implementation phases of the service-oriented design and development life cycle. The result of testing is a ‘healthy’ service-oriented application that performs well enough to satisfy the needs of its customers.

The most interesting type of testing for service implementations is *dynamic testing*, which consists of running the implementation and comparing its actual to its expected behaviour before it is deployed. If the actual behaviour differs from the expected behaviour, a defect has been found. In the context of services, dynamic testing is used to perform a variety of types of tests such as functional tests, performance and stress tests, assembly tests and interface tests.

Functional testing covers how well the system executes the functions it is expected to execute – including user commands, data manipulation, searches and business processes, and integration activities. Functional testing covers the obvious surface type of functions, as well as the back-end system operations, such as security, database transactions and how upgrades affect the system.

The focus of performance testing in service-oriented environments is monitoring the system online response times and transaction rates under peak workload conditions. It also involves load testing, which measures the system's ability to handle varied workloads. Performance testing is related to stress testing, which looks for errors produced by low resources or competition for resources. It is also related to volume testing, which subjects the software to larger and larger amounts of data to determine its point of failure.

The objective of interface testing is to ensure that any service developed to interface with other services functions properly outside of its surrounding process. Interface testing should be performed while testing the function that is affected, *e.g.*, an order management process calling an inventory service.

Finally, assembly testing ensures that all services function properly when assembled into business processes. It also verifies that services that interact and interoperate function properly when assembled as apart of business processes.

In addition to functional tests, performance and stress tests, assembly tests and interface tests, there are a variety of additional tests that may need to be performed during the service test phase. These include network congestion tests, security tests, installability tests, compatibility tests, usability tests and upgrade tests. Tests have to be conducted to ensure that service security requirements such as privacy, message integrity, authentication, authorisation and nonrepudiation are met.

5.6 The service construction phase

As WS become acceptable from industry, organisations realise that there are several intricate issues pertaining to the deployment aspects of revenue-generating WS. Service provisioning is central to operating revenue-generating WS between organisations. The provisioning requirements for WS impose serious implications for the development methodology of services. Service provisioning is a complex mixture of technical and business aspects for supporting service client activities and involves choices for service governance, service certification, service enrolment, service auditing, metering, billing and managing operations that control the behaviour of a service during its use. We provide an overview of the most salient features of service provisioning in what follows.

5.6.1 Service governance

The goal of service governance is to align the business strategy and imperatives of an enterprise with its IT initiatives (Mitra, 2005; Potts *et al.*, 2003). When applied to service-oriented applications service governance may involve reviews of internal development projects as well as external reviews from the perspective of the service providers, using results from gap analysis.

Typical issues for internal reviews include whether the right types of services have been selected, whether all requirements for new services have been identified and so forth. To this end service governance may use as input the findings of the business gap analysis subphase. Other internal review issues also include whether the use of a particular service within an application would conform to enterprise-specific or government-mandated privacy rules, whether service implementation does not compromise enterprise-specific intellectual property, and so on. To achieve its stated objectives and support an enterprise's business objectives on strategic, functional and operational levels, service governance provides a well-defined structure. It defines the rules, processes, metrics and organisational constructs needed for effective planning, decision-making, steering and control of the SOA engagement to meet the business requirements of an enterprise and its customers (Balzer, 2004).

Two different governance models are possible. These are central governance versus distributed governance.

With *central governance*, the governing body within an enterprise has representation from each service domain as well as from independent parties that do not have direct responsibility for any of the service domains. The central governance council reviews any additions or deletions to the list of services, along with changes to existing services, before authorising the implementation of such changes. Central governance suits an entire enterprise.

With *distributed governance* each business unit has autonomous control over how it provides the services within its own enterprise. This requires a functional service domain approach. A central governance committee can provide guidelines and standards to different teams. Distributed governance suits distributed teams better.

5.6.2 Service certification

To establish that a service possesses some desired property, we need to use knowledge to predict properties that an assembled application may attain. Certification depends on compositional reasoning (Bachmann *et al.*, 2000), which identifies which properties of services are material for predicting or achieving some end-system properties, such as performance, safety, scalability and so on, and how to predict the 'values' of end-system properties from service properties. These contractual specifications must be expressive enough to capture all of the properties imposed by frameworks that will lead to measurable end-system quality attributes.

5.6.3 Service metering and rating

This process requires service providers to come up with viable business cases that address factors such as service metering, rating and billing.

Service metering model

Use of a service by a client must be metered if the service provider requires usage-based billing. Then the service provider needs to audit the service as it is used and bill for it. This could typically be done on a periodic basis and requires that a metering and accounting model for the use of the service be established. The model could allow the establishment of a service contract for each new subscriber and tacking and billing for using the subscribed hosted services. To achieve this, the service-metering model could operate on the assumption that WS with a high degree of value are contracted via, for example, SLAs.

Service rating/billing model

Software organisations that are used to the traditional up-front license/ongoing maintenance pricing structure for software should come up with annuity-based pricing models for the WS they provide. The pricing (rating) model could determine subscriber rates based on subscription and usage events. For example, the pricing model could calculate charges for services based on the quality and precision of the service and on individual metering events based on a service-rating scheme. The billing model associates rating details with the correct client account. It provides adequate information to allow the retrieval and payment of billing details by the client and the correct disbursement of payments to the service provider's suppliers (who are in turn service providers offering wholesale services to the original provider).

5.6.4 Service billing strategies

Increasingly, business models for commercial WS provisioning will become a matter of concern to service providers. From the perspective of the service provider a complex trading WS is a commercialisable software commodity. For example, a service provider may decide to offer simple services (with no quality of service guarantee) for free, while it would charge a nominal fee for use of its complex (added-value) WS. With complex trading WS the quality of service plays a highly important role and the service is offered for a price. These types of services are very different from the selling of shrink-wrapped software components, in that payment should be on an execution basis for the delivery of the service, rather than on a one-off payment for an implementation of the software. For complex trading WS, the service provider may have different charging alternatives. These may include payment on a per use basis, payment on a subscription basis, payment on a leasing basis, lifetime services and free services.

5.7 The service deployment phase

Deployment means rolling out new processes to all the participants, including other enterprises, applications and other processes. The service-oriented development methodology promotes a separation between development and deployment activities, which are grouped into separate phases that can occur at different times, and that different individuals with different skills can perform. This yields a true separation of concerns, enabling developers to repurpose software components, services and business processes.

The tasks associated with the deployment phase of the WS development life cycle include the publication of the service interface and service implementation definition. Services are deployed at the service provider side according to the four service realisation options that we examined in Section 5.2.4.

5.8 The service execution phase

Execution means ensuring that the new process is carried out by all participants – people, other organisations, systems and other processes. During the execution phase, WS are fully deployed and operational. During this stage of the life cycle, a service requester can find the service definition and invoke all defined service operations. The runtime functions include static and dynamic binding, service interactions as a function of Simple Object Access Protocol (SOAP) serialisation/deserialisation and messaging and interactions with back-end legacy systems (if necessary).

5.9 The service management and monitoring phase

Services management considers consistent management of end-to-end WS. Such activities are the target of the Web Services Distributed Management (WSDM) specification. WSDM essentially defines a protocol for interoperability of management information and capabilities in a distributed environment via WS. WSDM focuses on two distinct tasks in its attempt to solve distributed system management problems (Kreger *et al.*, 2005). The first activity area, called Management Using Web Services (MUWS), addresses the use of WS technologies as the foundation of a modern distributed systems management framework. This includes using WS to facilitate interactions between managed resources and management applications. In particular, MUWS defines how to describe the manageability capabilities of managed resources using WSDL documents. Expressing capabilities enables more efficient discovery and introspection of resources since managers typically focus on a particular management task or domain, and therefore need to be able to easily and efficiently determine the relevant capabilities of a manageable resource (Papazoglou and van den Heuvel, 2005). In addition, WSDM addresses the specific requirements for managing WS themselves just like any other resource. This activity is called Management of Web Services (MOWS).

An integral part of service management is service version control. A version of the service interface is a specific instance of a service interface at a particular point in time that came into existence owing to a revision or a change. Currently, versioning has not been built into the WS architecture. The WSDL 1.1 specification does not address the issue of WS versioning. This means that in situations where the interface of a particular service needs to be changed with a new version, WSDL cannot convey the change to the service requester. If a developer makes a change to a service interface, all older requesters would fail, and the failure would be undetectable to the WS infrastructure. In the following we shall describe a relatively simple approach to services versioning.

There are two types of changes in a WSDL document: backwards-compatible and nonbackwards-compatible changes (Brown and Ellis, 2004). A backwards-compatible change to service is one in which a new WS can continue to interact with an old client without causing a failure. These changes include the addition of new operations, new data types, new optional fields to existing data types, and type expansion. In contrast to backwards-compatible changes, nonbackwards-compatible changes cause an old client to

fail when attempting to interact with a new WS. These include renaming an operation, changing the parameters (in data type or order) of an operation and changing the structure of a complex data type.

Backwards-compatible versions are relatively easy to handle. For backwards-compatible changes, the WSDL document can simply be updated in the repository from which it is made available to requesters, and the existing WS may be updated. Every new edition of a WSDL document should be stored in a version-control system, and XML comments should be used to indicate unique version IDs or a version history.

For nonbackwards-compatible changes, XML namespaces need to be used to clearly delineate the versions of a document that are compatible. Using the namespaces approach, developers can qualify editions of a WSDL document as belonging to a specific namespace that identifies the version. To ensure that the various editions of a WSDL document are unique, a simple naming scheme that appends a date or version stamp to the end of a namespace definition can be used. This may follow the general guidelines given by the W3C for XML namespace definitions in identifying schemas. It may consist of the company name, followed by a date stamp, followed by one (or more) identifier that specifically denotes the particular namespace.

Once the namespace is changed the designer must determine what to do with old service requesters. One option is to generate a failure on the server end if a request for an older namespace is received. Another common option for dealing with this problem is to employ a WS intermediary (a router, for instance, as part of the service container) that determines what to do with WS requests that come in for any particular namespace. The router could examine the date stamp on the namespace and then route requests from the older namespace to an older version of the WS, while routing requests from the newer namespace to the new version of the WS.

The service-monitoring phase concerns itself with service level measurement. Monitoring is the continuous and closed-loop procedure of measuring, monitoring, reporting and improving the quality of service of systems and applications delivered by service-oriented solutions. Service level monitoring is a disciplined methodology for establishing acceptable levels of service that address business objectives, processes and costs.

The service-monitoring phase targets continuous evaluation of service level objectives and performance. To achieve this objective service monitoring requires that a set of QoS metrics are gathered on the basis of SLAs, given that an SLA is an understanding of the expectation of service. In addition, workloads need to be monitored and the service weights for request queues might need to be readjusted. This allows a service provider to ensure that the promised performance level is being delivered, and to take appropriate actions to rectify noncompliance with an SLA, such as reprioritisation and reallocation of resources.

To determine whether an objective has been met, SLA QoS metrics are evaluated based on measurable data about a service performance – *e.g.*, response time, throughput, availability, and so on – during specified times, and periodic evaluations. SLAs include other observable objectives which are useful for service monitoring. These include compliance with differentiated service-level offerings, *i.e.*, providing differentiated QoS for various types of customers (gold, silver, bronze), individualised service-level offerings, and requests policing which ensures that the number requests per customer stay

within a predefined limit. All these also need to be monitored and assessed. A key aspect of defining measurable objectives is to set warning thresholds and alerts for compliance failures. This results in preemptively addressing issues before compliance failures occur. For instance, if the response time of a particular service is degrading then the step could be automatically routed to a backup service.

6 Outlook

In this paper, we have described an experimental methodology for service-oriented design and development. The methodology that we presented reflects an attempt to define a foundation of design and development principles that applies equally well to WS and business processes. The methodology takes into account a set of development models (*e.g.*, top-down, bottom-up and hybrid), stresses reliance on reference models, and considers several service realisation scenarios (including green-field development, outsourcing and legacy wrapping).

In contrast to traditional software development approaches, the methodology that we introduced in this article emphasises activities revolving around service provisioning, deployment, execution and monitoring. We believe that these activities will become increasingly important in the world of services as they contribute to the concept of *adaptive service capabilities*, where services and processes can continually morph to respond to environmental demands and changes without compromising on operational and financial efficiencies. In this way, business processes could be analysed in detail instantaneously – discovering and selecting suitable external services, detecting problems in the service interactions, searching for possible alternative solutions, monitoring execution step by step, upgrading and versioning themselves, and so on.

Service adaptivity is particularly useful for integrated supply chains, as it implies that an integrated supply chain solution can leverage collaborative, monitoring and control abilities to manage product variability and successfully exploit the benefits of Available-To-Promise (ATP) capabilities. For example, consider the case where an enterprise receives a direct request from its customer order-entry service. An order-promising service routes this request instantaneously to all sites that could fulfil the order. Frequently there are multiple, hierarchically ordered partners with facilities in different geographic regions. The ATP service for available and planned inventory is then checked against the date requested by the customer and the appropriate quantities. If necessary, substitute choices are offered. The ATP results are then sent to a transportation-planning business process of a logistics service provider to determine transportation time and delivery dates. The results are subsequently relayed to the order-promising service, which selects the fulfilment site and responds to the customer-order service for approval. Order acceptance is then propagated back through the system, driving the acceptances. However, if the material is not available, the order-promising service can use the capable-to-promise functionality to contact a production-scheduling service and establish a date for the products promised. All these steps involve conversation between processes that span enterprises, with customised alerts set up across the network to track exceptions and provide manual intervention if necessary.

We intend to further strengthen and refine the approach by conducting several real-life case studies in different sectors, from which experience will be gained and more design concerns may be derived. In addition, we plan to develop an integrated toolset to effectively support the methodology.

References

Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2004) *Web Services: Concepts, Architectures and Applications*, Heidelberg: Springer.

Andrews, T., *et al.* (2003) *Business Process Execution Language for Web Services, Version 1.1*.

Arsanjani, A. (2004) 'Service-oriented modeling and architecture', *IBM Developerworks*, Novemeber, http://www-106 ibm. com/developerworks/library/ws-soa-design1/.

Atkinson, B., *et al.* (2002) *Web Services Security (WS-Security)*, International Business Machines Corporation, MicrosoftCorporation, VeriSign, Inc., Version 1.0, April.

Bachmann, F., *et al.* (2000) 'Technical concepts of component-based software engineering', *Technical Report*, 2nd ed., Carnegie-Mellon University, CMU/SEI-2000-TR-008 ESC-TR-2000-007, May.

Balzer, Y. (2004) 'Strong governance principles ensure a successful outcome', *IBM Developerworks*, July, http://www-106.ibm.com/developerworks/library/ws-improvesoa/.

Brittenham, P. (2001) *Web-Services Development Concepts*, IBM Software Group, May, http://www-06.ibm.com/software/solutions/webservices/.

Brown, A., *et al.* (2005) 'SOA development using the IBM rational software development platform: a practical guide', *Rational Software*, September.

Brown, K. and Ellis, M. (2004) 'Best practices for web services versioning', *IBM Developerworks*, January, http://www128.ibm.com/developerworks/webservices/library/w s-version/.

Chappell, D. (2004) *Enterprise Services Bus*, O'Reilly.

Chinnici, R., Gudgin, M., Moreau, J-J., Schlimmer, J. and Weerarana, S. (2004) *Web Services Description Language (WSDL) Version 2.0*, March, w3c.org 4.

El Sawy, O. (2001) *Redesigning Enterprise Processes for e-Business*, McGraw-Hill.

Harmon, P. (2003) 'Second generation business process methodologies', *Business Process Trends*, May, Vol. 1, No. 5.

Herzum, P. and Sims, O. (2000) *Business Component Factory*, J. Wiley and Sons Inc.

Johnston, S. (2005) 'Modelling service-oriented solutions', *IBM Developerworks*, July, http://www128.ibm.com/developerworks/rational/library/johnston/.

Kreger, H., *et al.* (2005) *Management Using Web Services: A Proposed Architecture and Roadmap*, IBM, HP and Computer Associates, June.

Kruchten, P. (2004) *Rational Unified Process – An Introduction*, 3rd ed., Addison-Wesley.

Mitra, T. (2005) 'A case for SOA governance', *IBM Developerworks*, August, http:// www-106.ibm.com/developerworks/webservices/library/ws-soa-govern/index.html.

Papazoglou, M.P. and Georgakapoulos, G. (2003) 'Introduction to the special issue about service-oriented computing', *CACM*, October, Vol. 46, No. 10, pp.24–29.

Papazoglou, M.P. and van den Heuvel, W.J. (2005) 'Web services management: a survey', *IEEE Internet Computing*, November–December.

Papazoglou, M.P. (2006) *Principles and Foundations of Web Services: A Holistic View*, Addison-Wesley, to appear: 2006.

Potts, M., *et al.* (2003) 'Web service manageability – specification (WS-manageability)', *OASIS*, September.

Royce, W. (1998) *Software Project Management: A Unified Framework*, Addison-Wesley.

Rational Unified Process (RUP) (2001) 'Rational Software Corporation "Rational Unified Process": Best practices for software development teams', *Technical Paper TP026B*, November, Rev. 11/01, http://www.therationaledge.com.

White, S.A. (2004) 'Introduction to BPMN', *Business Process Trends*, July, www.bptrends.com.